ARTIFICIAL NEURAL NETWORK BASED ADAPTIVE CONTROLLER FOR DC MOTORS

WIDANALAGE RAVIPRASAD DE MEL
B.Sc.Eng., University of Moratuwa M.Sc., University of Peradeniya

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF ENGINEERING

DEPARTMENT OF MECHANICAL ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2003

Acknowledgements

I wish to express my sincere gratitude to my supervisor, Professor Poo Aun Neow for his invaluable guidance, advice and support throughout this thesis project. Professor Poo's success and enthusiasms in research helped me to arouse my interest in various aspects of control and mechatronics engineering. I also wish to thank Professor Clarence W. de Silva of the Department of Mechanical Engineering at the University of British Columbia for introducing me to Professor Poo and for his fine advice.

I deeply appreciate the scholarship awarded to me to do this research degree by the Sri Lankan Government under the Science and Technology Personnel Development Project. My special thanks to Mr. P.D. Sarath Chandra, head of the Mechanical Engineering Department at the Open University of Sri Lanka for nominating me for the scholarship and for the various advice given to me during my career.

My friendly thanks and best wishes go to my fun-loving fellow postgraduate students of the Control and Mechatronics Laboratory, National University of Singapore, for providing a conducive environment to work. The assistance given by the technical staff of the Control Division is gratefully acknowledged.

I also like to thank my wife Maheeka, my parents and my sister for their love, support and encouragement during the long period of study from my childhood and for taking other burdens on behalf of me. My special thanks go to my son Geeth, for understanding and waiting patiently while I was away from home at the time he needed the father's safeguard most.

Table of Contents

Summary

This thesis studies the development, implementation, and performance of an on-line self-tuning artificial neural network (ANN) based adaptive speed controller for a permanent magnet dc motor. For more accurate speed control, an on-line training algorithm with an adaptive learning rate is introduced, rather than using fixed weights and biases for the ANN. Both analytical and practical details of the development and implementation of the ANN based adaptive controller techniques are systematically presented. The complete system is implemented in real time using a host-target prototyping environment and a laboratory PM (permanent-magnet) DC motor. To validate its efficiency, the performance of the proposed ANN-based adaptive controller was compared with proportional-integral-derivative (PID) and proportional-integral (PI)-controller-based PM DC motor drive systems under different operating conditions. The experimental results show that the ANN based adaptive controller is robust, accurate, and insensitive to parameter variations and load disturbances.

List of Figures

Chapter 1

INTRODUCTION

The evolution of living organisms exhibits the key characteristic of adaptation to their environment. They attempt to keep their physiological equilibrium to face the changes in the environmental surroundings. In day-to-day usage, adapt means to adjust to conform to new circumstances. In control engineering, an adaptive controller is a regulator that can modify its behavior in response to changes in the dynamics of the process and the disturbances. The history of adaptive controls runs way back to the early fifties when extensive research was carried out in connection with the design of autopilots for high performance aircrafts. A dynamic controller of this type, as opposed to a linear feedback controller, is required to sustain the dynamic performance of the aircraft for the entire range of its operating conditions. The adaptation feature gives the robustness to the controller in highly nonlinear, time varying systems. If an artificial neural network is used to mimic the adaptive feature of the controller, which roughly resembles the biological brain structure, using the knowledge of mathematical models acquired through learning, we would be able to enhance the adaptability of the controller.

1.1 Motivation

Recent developments in microprocessors, magnetic materials, semiconductor technology, and mechatronics provide a wide scope of applications of high-

performance electric motors in various industrial processes. In high-performance motor drive applications involving mechatronics, such as robotics, rolling mills, machine tools, etc., an accurate speed and/or position control is of critical importance. Although relatively expensive, DC motors are still widely used in such applications because of their reliability and ease of control due to the decoupled nature of the field and armature magneto motive forces (MMF's).

In high-performance drive applications like robots and disc drives, the control of a DC motor demands special attention because it must meet the criteria of fast response, quick recovery of speed from load impact, precise trajectory tracing and insensitivity to parameter variations. Conventional designs for robust control are often associated with constant gain controllers, such as proportional integral (PI) or proportional integral derivative (PID), which stabilize a class of linear systems over a small range of system parameter variations. Moreover, these types of systems need accurate mathematical models to describe the system dynamics for proper controller design. These are often quite difficult to obtain in practical situations.

In recent years, many adaptive control techniques, such as model reference adaptive control (MRAC), sliding mode control (SMC), variable structure control, and self-tuning regulators have been introduced in modern drive systems. These conventional adaptive control techniques are usually based on system model parameters. The unavailability of an accurate system dynamic model often leads to a cumbersome design approach. In addition, most of the adaptive control techniques for nonlinear systems are often associated with linearizing the model for a specific operating time interval and applying linear control theories. This introduces

considerable errors because of the linearization of the nonlinear model. Real-time implementation is often difficult and sometimes not feasible because of the use of a large number of parameters in these adaptive schemes.

Recently, multilayer feedforward neural networks (FFNN's) have proven extremely useful in pattern recognition, image processing, and speech recognition. These networks are also receiving wide attention in control applications. When an artificial neural network (ANN) is used as a motor controller in real time, it can tune itself through on-line training and instruct the motor drive system to perform according to the desired way. Thus, the inherent parallel and distributed architecture of an ANN can be successfully used for the control of an electric motor. The ANN can provide a nonlinear mapping between inputs and outputs of an electric drive system, without the knowledge of any predetermined model. Therefore, the use of an ANN in high-performance motor drives can make the system robust, efficient, and immune to undesired operating conditions.

Relatively fewer works have been reported in the literature about the successful control of DC motors using ANN as an adaptive controller. Therefore there is a need to develop an efficient on-line self-tuning ANN-based DC motor controller, which can exhibit the adaptive feature.

1.1.1 Goals of the Research

The main objective of the research reported in this thesis is to study the effectiveness of knowledge based adaptive control with particular emphasis on DC motor control. ANN is used for expressing the knowledge base-adaptation in the controller. The developed techniques will be tested and experimented. These experimental results are compared with traditional control techniques, using software and hardware.

1.1.2 Scope of the Study

This study covers the investigation of adaptive control techniques in DC motor control. The development of this adaptive control is incorporated in an ANN controller with digital feedback. In order to verify and gain insight into the developed adaptive controller, computer simulation studies are carried out using Mathwork's MATLAB ® and Simulink ®. The specific ANN-based adaptive controller technique is then prototyped in real-time by using the xPC target ® in MATLAB®. The software programming is carried in the host-target computer setup. The hardware interfacing work is carried out by establishing communication between the DC motor and the target computer with the help of an I/O card. The ANN-based controller model is built in the host computer using MATLAB ® and coded in C with the aid of Watcom C_C^{++}, and then down loaded to the target PC. After the overall implementation setup is established and tested for proper functioning, performance evaluation of the ANN-based adaptive controller is performed through extensive experimentation.

1.2 Literature review

From the very beginning, it has been realized by systems theorists that most real world dynamical systems are nonlinear. However, linearisations of such systems around the equilibrium states yield linear models, which are mathematically obedient. In particular, based on the superposition principle, the output of the system can be computed for any arbitrary input, and alternately, in control problems, the input, which optimizes the output in some sense, can also be determined with relative ease. In most of the adaptive control problems, where the plant parameters are assumed to be unknown, the fact that the latter occur linearly makes the estimation procedure straightforward. The fact that most nonlinear systems thus far could be approximated satisfactorily by linear models in their normal ranges of operation has made them attractive in practical contexts as well. It is this combined effect of ease of analysis and practical applicability that accounts for the great success of linear models and has made them the subject of intensive study for over four decades. In recent years, a rapidly advancing technology and a competitive market have required systems to operate in many cases in regions in the state space where linear approximations are no longer satisfactory. To cope with such nonlinear problems, research has been underway on their identification and control using artificial neural networks based entirely on measured inputs and outputs.

The term artificial neural networks (ANN's) have come to mean any architecture that has massively parallel interconnection of simple processors. From a theoretical point of view, a neural network can be considered as conveniently a parameterized class of nonlinear maps. During the 1980's and early 1990's conclusive

proofs were given by numerous authors, that multi layer feedforward networks are capable of representing any nonlinear continuous functions to any degree of required accuracy provided that the networks are sufficiently large and properly trained. This phenomenon in ANN has gained wide attention in control applications.

This inherent parallel and distributed architecture of ANN can be successfully used for control of PM DC motor drive system. Some useful works on the speed control of DC motor drives using ANN based speed controllers were reported [13], [14], [2], [5], [4]. In Weerasooriya and Sharkawi [13-14] a DC motor was successfully controlled using an ANN, which has a capability of capturing the unknown, time invariant, nonlinear operating characteristics of the DC motor. However their works are primarily based on an off-line trained ANN with indirect model reference adaptive technique (MRAC). Due to the absence of on-line training of the ANN, the speed control is not totally satisfactory. This is because under unknown operating conditions, that are not considered during the off-line ANN training process, the ANN controller does not perform well. The ANN based adaptive controller for a permanent magnet DC motor by El-Khouly and others [2] incorporate on-line updating. In their work, they found that while they were able to obtain good control performance, sometime the on-line updating of the weights become unstable resulting in the DC motor running away. The system they used, was different from the inverse dynamic model, the reference speed was arbitrarily taken as one of the inputs of the ANN structure, resulting in the driver system suffering from the problem of instability.

Hoque, Zaman and, Rahman [4], [5] have reported work on a real-time implementation of an ANN based control of a PM DC motor drive. In their works a PM DC motor drive system with ANN speed controller is designed. A multi layer ANN structure with one feedback loop is adopted in order to achieve an adaptive speed control over a wide operating range with load and parameter variations. This arrangement involves both off-line and on-line weight and bias updating for the ANN using the back-propagation algorithm. Here the stability over a wide range of operating points was obtained by using an ANN structure with feedback loop. Although the drive system stability has been improved, the evaluated system responses have considerable amounts of speed overshooting under some operating conditions. This is because the learning rate is not adaptive during the on-line weights and biases updating.

Narendra and Mukhopadhyay [10] in their work introduced two classes of models which are approximations to the NARMA (The NARMA model is an exact representation of the input–output behavior of finite-dimensional nonlinear discrete-time dynamical systems in a neighborhood of the equilibrium state) model, and which are linear in the control input. Their extensive simulation studies have shown that the neural controllers designed using the proposed approximate models perform very well, and in many cases even better than an approximate controller designed using the exact NARMA model.

The work reported by Rubaai and Kotaru in their paper [12] tackles the problem in a more general sense. No attempt is made to linearize the dynamics of the motor/load, preserving the fidelity of the model completely. The motor/load dynamics

are modeled online and controlled using a dynamic backpropagation (DBP) neural network. Two control topologies are considered. No a priori knowledge of the load dynamics is assumed in either topology, while the second topology also assumes no knowledge of the motor parameters. An adaptive learning algorithm that utilizes an adaptive learning rate for training the neural network is introduced. They have presented some comparison between the results obtained using the DBP algorithm and those obtained using the learning rate adaptation and reveals that the latter is much more efficient.

After studying the past work done by many researchers regarding the ANN-based DC motor controllers, if ones wants to design an efficient and stable on-line self –tuning ANN-based DC motor controller, one has to introduce an adaptive learning rate feature. Therefore the work presented in this thesis is based on a new speed control strategy of a PM DC motor incorporating an on-line weights and biases updating feature of the ANN. The ANN architecture is based on the inverse dynamic model of the nonlinear drive system. To enhance the robustness, which is an important criterion of a high-performance drive, a unique feature of adaptive learning rate is also introduced.

1.3 Contributions and Organization of the Thesis

The main contribution of this thesis is the development an Artificial Neural Network based adaptive controller for a permanent magnet direct current motor. It has been implemented in real time. These performances are compared with conventional

Proportional Integral (PI) and Proportional Integral Derivative (PID) control techniques. The relative advantages and disadvantages of the controller are identifiable. The proposed ANN based adaptive controller system applied to the PM DC motor is found to be robust, efficient and easy to implement.

The rest of the chapters present details on the research and development as given above. Chapter Two presents the theoretical development of the ANN based adaptive controller, starting with the conventional adaptive approach, followed by two digital servo controllers, which are commonly used in servo controller systems. Then it discusses the dynamics of motor drive systems followed by the ANN model for the PM DC motor. This chapter also discusses how the FFNN structure is used to develop the ANN based adaptive controller. Chapter Three focuses on the construction and training of the ANN controller. It discusses the off-line and on-line training of the ANN structure and the updating of the weights and biases. Chapter Four gives the details of practical implementation in real time. It further discusses the system architecture hardware interfacing and software architecture used in the research. In Chapter Five the experiment results and outcomes are presented. It also gives the results of the comparison of the ANN structure with PI, PID controllers. Chapter Six is the final and concluding chapter, which summarises the overall research and presents suggestions and possible directions of further research in the area.

Chapter 2

THEORETICAL DEVELOPMENT

From the beginning of systematic automatic controller design there has been the problem of finding a proper controller structure and the controller parameters for a given process. The main difficulty that comes into sight is the need of the controller to be very well tuned for the whole range of its operating points rather than for one particular operating point. To overcome these circumstances, adaptive controllers were developed in the nineteen forties. Between nineteen sixties and nineteen seventies many fundamental areas in control theory were developed which later proved to be significant for the design of adaptive control systems, e.g. state space and stability theory.

2.1 Adaptive control

Adaptive controllers are characterized by their ability to gather information about the parameters of a process during actual control and by their ability to make changes to their control laws accordingly based on the information gathered. Most adaptive controllers can be divided into two main classes: feedforward adaptive controllers and feedback adaptive controllers.

2.1.1 Feedforward Adaptive controllers.

These systems are based on the fact that the changing properties of the plant can be grasped by measurement of signals acting on the process. It is know-how that the controller must be changed depending on these signals. The feedforward adaptation system can be realized as shown in Fig. 2.1.

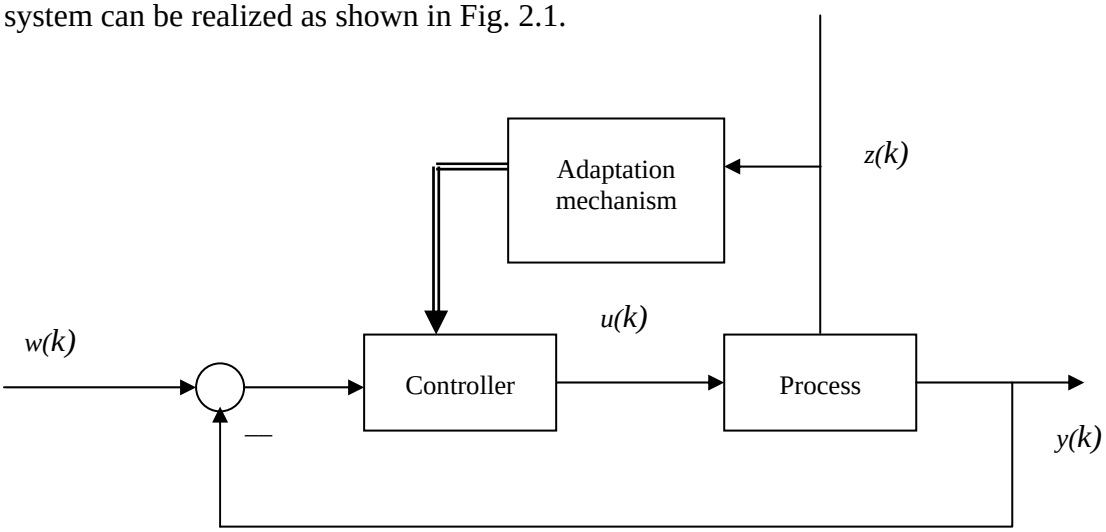

Figure 2.1 Feedforward adaptive control (open-loop adaptation).

A special feature of this controller is that there is no feedback of 'inner' closed-loop signals to adapt the controller parameters. In Fig. 2.1, the disturbance input (*z(k)*) is measured and the adaptive mechanism changes the parameters of the controller in such away as to maintain good control performance. One advantage of feedforward adaptive control is that fast reaction to process changes can be achieved because the process behavior could be anticipated and need not be identified with measurable process input and output signals. There are some disadvantages in this system. They are neglect of effects based on unmeasured signals or disturbances, unpredictable changes of the process behavior and the amount of parameter storage that may be necessary to accommodate many operating conditions and the limitations to slow processes and parameter changes.

2.1.2 Feedback adaptive controllers

Feedback adaptive controllers are used when the process behavior changes cannot be determined directly by measurement of external process signals. The basic structure of the feedback adaptive controller is shown in Fig. 2.2. These controllers are characterized by the following three factors. First, the changing properties of the process or its signals can be observed by the measurement of different internal control loop signals. Secondly, in addition to the basic control loop feedback, the adaptation mechanism results in an additional feedback level. Thirdly, the closed-loop signal flow path yields a nonlinear second feedback level.

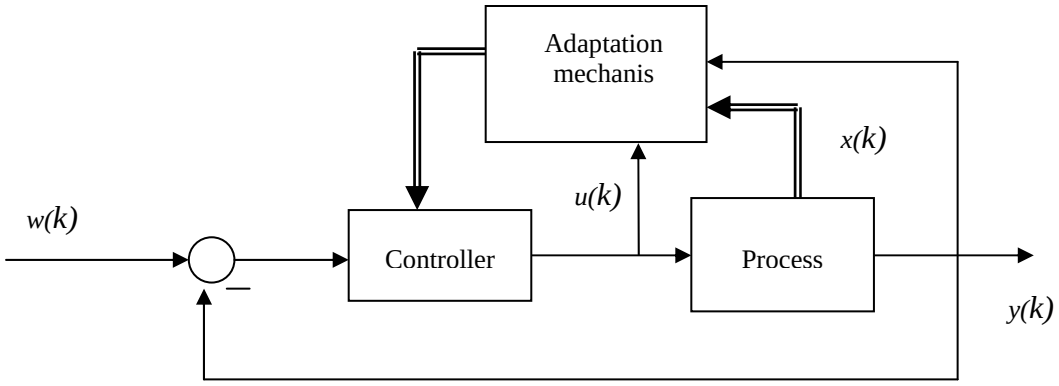

Figure 2.2 Feedback adaptive control (closed-loop adaptation).

2.2 Digital Servo Controllers

Other than the ANN controller two types of conventional feedback controllers are used in the present study. One is a Proportional-Integral (PI) controller and the other is a Proportional-Integral-Derivative (PID) controller. Both these servo controllers are used for comparison purposes with the Artificial Neural Network (ANN) based controller. At implementation the controllers were built using a host-target

prototyping environment with a compatible data acquisition board. In this study a permanent magnet (PM) DC motor is adopted as the plant.

2.2.1 PI Controller

The idealized equation of a proportional-integral (PI) controller is

$$u(t) = K\left[e(t) + \frac{1}{T_i}\int_o^t e(t)dt \right] \tag{2.1}$$

in which K is the gain, T_i is the integral time and $e(t)$ is the feedback error; i.e., $e(t) = r(t) - y(t)$. Where $r(t)$ and $y(t)$ are reference input and the plant output respectively The equivalent transfer function in the s-domain is given by

$$U(s) = \left[K\left(1 + \frac{1}{T_i s} \right) \right] E(s) \tag{2.2}$$

For digital control, Equation (2.2) is transformed into its discrete-time (z-domain) equivalent, as given by

$$U(z) = \left[K_P + \frac{K_I}{1 - z^{-1}} \right] E(z) \tag{2.3}$$

or, in velocity form,

$$U(z) = -K_P Y(z) + K_I \frac{E(z)}{1 - z^{-1}} \tag{2.4}$$

where $$K_P = K - \frac{KT_s}{2T_i}, \tag{2.5}$$

$$K_I = \frac{KT_s}{T_i}, \tag{2.6}$$

and T_s is the sampling interval.

2.2.2 PID Controller

The idealized equation of a proportional-integral-derivative (PID) controller is

$$u(t) = K\left[e(t) + \frac{1}{T_i}\int_o^t e(t)dt + T_d \frac{de(t)}{dt} \right] \tag{2.7}$$

in which K is the gain, T_i is the integral time, T_d is the derivative time, and $e(t)$ is the feedback error; i.e., $e(t) = r(t) - y(t)$. The equivalent transfer function in the s-domain is given by

$$U(s) = \left[K\left(1 + \frac{1}{T_i s} + T_d s \right) \right] E(s) \tag{2.8}$$

For digital control, equation (2.8) is transformed into its discrete-time (z-domain) equivalent, as given by

$$U(z) = \left[K_P + \frac{K_I}{1 - z^{-1}} + K_D(1 - z^{-1}) \right] E(z) \tag{2.9}$$

or, in velocity form,

$$U(z) = -K_P Y(z) + K_I \frac{E(z)}{1 - z^{-1}} - K_D(1 - z^{-1})Y(z) \tag{2.10}$$

where $$K_P = K - \frac{KT_s}{2T_i}, \tag{2.11}$$

$$K_I = \frac{KT_s}{T_i}, \tag{2.12}$$

$$K_D = \frac{KT_d}{T_i} \tag{2.13}$$

and T_s is the sampling interval.

2.3 Adaptive control using ANN

Human thinking has both logical and intuitive or subjective sides. The logical side has been developed and used, resulting in present advanced von Neumann type computers and expert systems, both constituting the hard computing domain. However, it is found that hard computing cannot give the solution of real, very complex and nonlinear systems by itself. In order to cope with this difficulty, the intuitive and subjective thinking of the human mind was exploited, resulting in soft computing approaches that include neural networks and fuzzy logic based reasoning.

Recent applications in different domains proved that superior results could be obtained using artificial neural networks. The ANN provides a nonlinear mapping between inputs and outputs of an electric drive system, without the knowledge of any predetermined model. Therefore, the use of an ANN in adaptive control can make the systems robust and efficient.

In the proposed work, an adaptive speed control strategy for a PM DC motor is used incorporating an on-line updating of the weights and biases of the ANN controller. The ANN architecture is based on the inverse dynamic model of the nonlinear drive system. To enhance the robustness, which is an important criterion of a high-performance drive, a unique feature of adaptive learning rate is also used.

2.4 DC Motor Drive System Dynamics

Although it is not necessary to obtain a motor model if the ANN is used in the motor control scheme, it is important doing so from the analytical point of view, in order to set up the groundwork of the ANN structure.

The PM DC motor dynamics are described by the following equations

$$v_a(t) = R_a i_a(t) + L_a \frac{di_a(t)}{dt} + e_b(t) \tag{2.14}$$

$$e_b(t) = K_E \omega_r(t) \tag{2.15}$$

$$T_e(t) = K_T i_a(t) \tag{2.16}$$

$$T_e(t) = J \frac{d\omega_r(t)}{dt} + B\omega_r(t) + T_l(t) + T_F \tag{2.17}$$

Where

$v_a(t)$ - Motor terminal voltage

$e_b(t)$ - Motor back EMF

$i_a(t)$ - Armature current

$\omega_r(t)$ - Motor rotating speed

R_a - Armature resistance

L_a - Armature inductance

K_E - Motor back EMF constant

K_T - Torque constant

$T_e(t)$ - Developed torque

$T_l(t)$ - Load torque

T_F - Frictional torque

J - Inertia constants

B - Viscous constants

The load torque, $T_l(t)$, can be expressed as

$$T_l(t) = \Psi(\omega_r(t)) \tag{2.18}$$

where the function $\Psi(\omega_r(t))$ depends on the nature of the load. The exact functional expression of $\Psi(\omega_r(t))$ is assumed to be unknown.

In order to derive training data for the ANN and to apply the control algorithms, a discrete-time DC motor model is required. Let's assume the load torque $T_l(t)$ of Equation (2.18) to be non-linear and of the form

$$T_l(t) = v\omega_r^2(t)\left[sign\{\omega_r(t)\}\right] \tag{2.19}$$

where v is a constant used for modeling the nonlinear mechanical load. Although the load expressed by (2.19) is assumed as a fan or propeller type for modeling purposes, in real life, it is uncertain and usually has unknown nonlinear mechanical characteristics. To make the control task easier, the PM DC motor drive system can be expressed as a single-input single-output (SISO) system by combining (2.14)–(2.17), giving

$$L_a J \frac{d^2\omega_r(t)}{dt^2} + (R_a J + L_a B)\frac{d\omega_r(t)}{dt} + (R_a B + K_E K_T)\omega_r(t) + L_a \frac{dT_l(t)}{dt}$$

$$+ R_a \{T_l(t) + T_F\} - K_T v_a(t) = 0 \qquad (2.20)$$

The discrete-time model is derived by combining equations (2.19) and (2.20) and then replacing all continuous differentials with finite differences. The resulting state space equation is

$$\omega_r(n+1) = K_1\omega_r(n) + K_2\omega_r(n-1) + K_3[sign\{\omega_r(n)\}]\omega_r^2(n)$$

$$+ K_4[sign\{\omega_r(n)\}]\omega_r^2(n-1) + K_5 v_a(n) + K_6 \qquad (2.21)$$

where K_1, K_2, K_3, K_4, K_5 and K_6 are constants and can be expressed in terms of the motor parameters. The expressions for the above constants are given in Appendix A. Equation (2.21) can be further modified to obtain the inverse dynamic model of the drive system as

$$v_c(n) = f[\omega_r(n+1), \omega_r(n), \omega_r(n-1)] \qquad (2.22)$$

where $v_c(n)$ is the control voltage of a power converter and is linearly proportional to the terminal voltage $v_a(n)$. The right-hand side of (2.22) is a nonlinear function of the speed ω_r and is given by

$$f[\omega_r(n+1), \omega_r(n), \omega_r(n-1)] =$$

$$[\omega_r(n+1) - K_1\omega_r(n) - K_2\omega_r(n-1) - K_3[sign\{\omega_r(n)\}]\omega_r^2(n)$$

$$- K_4[sign\{\omega_r(n)\}]\omega_r^2(n-1) - K_6]/K_5 \qquad (2.23)$$

The purpose of using the ANN is to map the nonlinear relationship between the terminal voltage $v_c(n)$ and the speed $\omega_r(n)$ of the DC motor according to (2.22). Derivation of (2.22) allows the structure of the ANN required for speed control of the PM DC motor drive to be estimated.

2.5 ANN Structure for the Controller

2.5.1 Feedforward neural network structure (FFNN)

First the general structure of the FFNN is discussed before designing the problem-specific ANN. The general architecture of the FFNN is shown in Fig. 2.3. The network consists of one input layer and one or more hidden layers, followed by an output layer. Each layer consists of a number of neurons. Each neuron has two functions. The first is to sum up all the outputs from the previous layers multiplied by the corresponding connecting weights. The second function is to perform a nonlinear (e.g., sigmoidal) or a linear function on this sum.

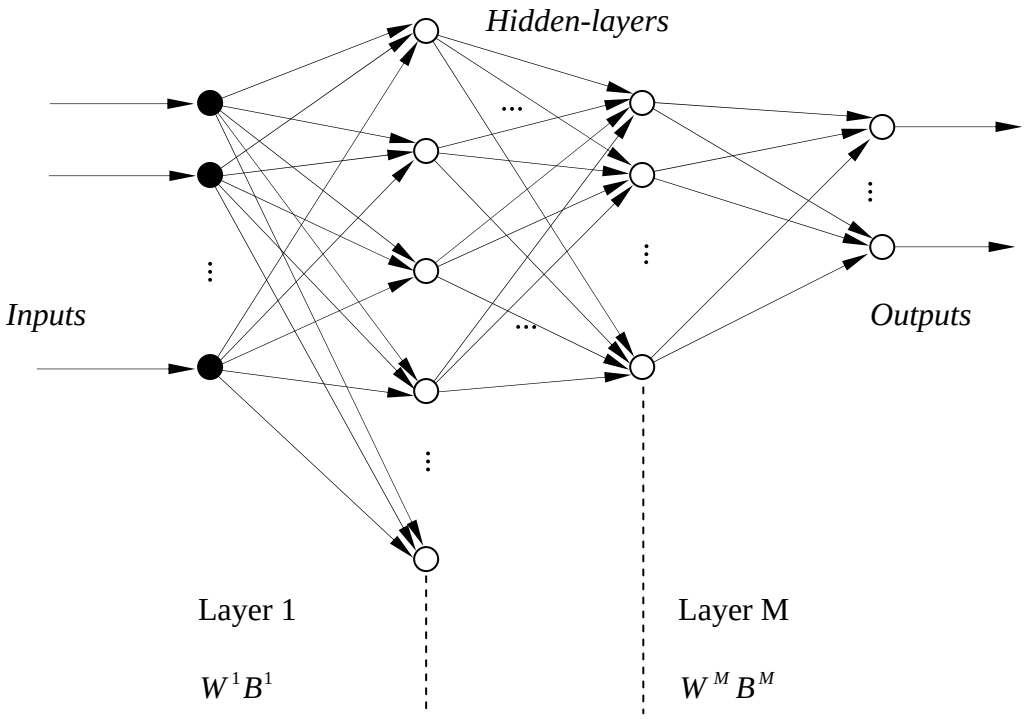

Figure 2.3 A general FFNN structure.

The fundamental equations, which describe the inputs and outputs of the network, can be formulated as follows[4].

The net input of the j^{th} neuron of the hidden layer at the time instant n is given by

$$S_j^h(n) = \sum_{i=1}^{N} W_{ij}^h(n) I_i(n) \tag{2.24}$$

where W_{ij}^h is the connecting weight between the i^{th} neuron at the input layer and the j^{th} neuron at the hidden layer, I_i is the i^{th} input, and N is the number of inputs. The output from the j^{th} neuron from the hidden layer at n^{th} instant is given as

$$O_j^h(n) = f^h[S_j^h(n) + B_j^h(n)] \tag{2.25}$$

where B_j^h is the bias of the j^{th} neuron and f^h is the nonlinear activation function acting at the output of each neuron in the hidden layer. The activation function used in generally tan sigmoidal or log sigmoidal, which are defined as

$$tansig(x) = \frac{1 - e^{-2x}}{1 + e^{2x}} \tag{2.26}$$

$$logsig(x) = \frac{1}{1 + e^{-x}} \tag{2.27}$$

The net input of the k^{th} neuron of the output layer at time instant n is given by

$$S_k^o(n) = \sum_{j=1}^{M} W_{jk}^o(n) O_j^h(n) \tag{2.28}$$

where M is the number of neurons in the hidden layer and $W_{jk}^o(n)$ is the connecting weight between the j^{th} neuron at the hidden layer and the k^{th} neuron at the output layer. The output from the k^{th} neuron at the output layer at time instant n is given by

$$O_k^o(n) = f^o[S_k^o(n) + B_k^o(n)] \tag{2.29}$$

where f^0 is an activation function and $B_k^o(n)$ is the bias of the k^{th} neuron at the output layer.

2.5.2 Artificial Neural Network Structure for motor drive

The most important task of designing an ANN controller is to determine the input(s) and output(s). The dynamics of the PM DC motor drive described in (2.22) basically state the inputs and output of the ANN needed for the control of the PM DC motor drive system under consideration. The left-hand side of (2.23) gives the required inputs of the ANN structure, i.e., $\omega_r(n+1), \omega_r(n),$ and $\omega_r(n-1)$, three consecutive values of speed, and the corresponding output target is the control voltage $v_c(n)$. The number of hidden layers and the number of neurons in the hidden layer are chosen by trial and error, In deciding in the number of neurons, it is noted that the smaller the number, the better it is in terms of both size of memory required and computational load. On the other hand, too small number may result in an ANN that may not be able to accurately map the function required. The ANN structure used for the PM DC motor drive is shown in Fig. 2.4.

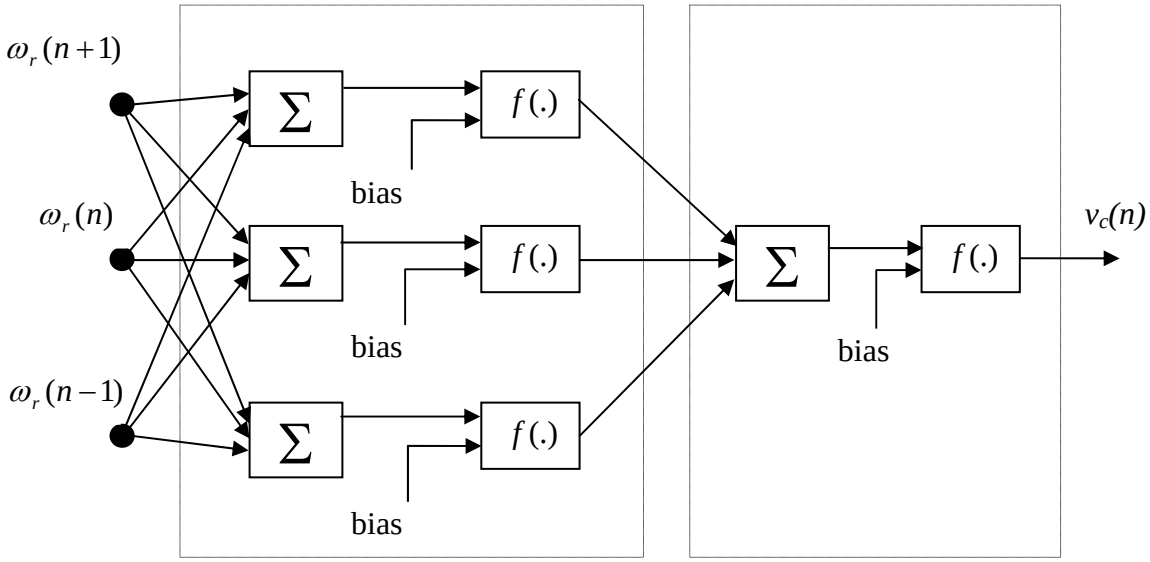

Figure 2.4 ANN structure for PM DC motor drive.

The activation functions used in the hidden and output layers are log sigmoid and tansigmoid respectively. After this basic design of the ANN structure is done, the next step is to establish the weights and biases of the ANN through training to obtain the specific target with the given inputs. Here, the back-propagation training algorithm is used for this purpose, which is based on the principle of minimization of the cost function of the error between the outputs and the target of the FFNN [4]. Training of the network can be done either off-line or on-line, depending on the application. If the weights and biases of the network are determined through off-line training only, then extensive training has to be performed, taking into consideration almost all operating conditions of the system, which is practically impossible for the control of a PM DC motor. Therefore, a combination of off-line and on-line training is used here.

2.6 Summary

The theoretical development of a neural network based adaptive controller for a permanent magnet direct current motor was presented in this chapter. First the concept of adaptive control was introduced, and then the main classes of adaptive controllers were discussed. Two types of conventional digital servo controllers, the proportional-integral (PI) and proportional-integral-derivative (PID) controllers, were also discussed. Then the motor model for the PM DC motor was explained by using the motor drive system dynamics. Although it is not absolutely necessary to derive a motor model if an Artificial Neural Network is used in the motor control scheme, having a model helps in the design of a suitable ANN structure to be used. After that adaptive control using ANN were discussed. Next, the structure of FFNN was explained and finally the ANN structure for the motor drive control was introduced.

Chapter 3

ANN BASED ADAPTIVE CONTROLLER

As mentioned in the previous chapter, one of the native characteristics of the ANN is its capability to map any non-linear relation between input and output through training and without the need for knowledge of any predetermined model. Exploiting this property a multi-layer feed forward neural network structure is used here. This structure, which has an adaptive capability, is used to control the speed of a PM DC motor.

3.1 ANN Structure for System Identification and Control

The objective of a speed control system for a DC motor is to produce the appropriate control signal, in this case the terminal voltage for the DC motor, so that the motor can track the reference speed $\omega_{ref}(n)$. At each sampling instant a control voltage $v_c^*(n)$ for the PM DC motor is generated by the ANN structure, which is fed to a power amplifier circuit as shown in the Fig. 3.1. ANN1 and ANN2 shown in Fig. 3.1 have the identical set of weights and biases but two different sets of inputs and outputs. In the present study, the ANN structure comprising one hidden layer having three neurons with one neuron in the output layer gives satisfactory results. The output of the power amplifier is applied to the terminal of the motor. More details regarding the practical setup will be extensively explained in the next chapter.

The reference speed trajectory is selected using a second-order reference model that makes the system asymptotically stable [13]. The reference model is

described by the following equation:

$$\omega_{ref}(n+1) = a_1\omega_{ref}(n) + a_2\omega_{ref}(n-1) + r(n) \tag{3.1}$$

where a_1 and a_2 are constants chosen for a reference trajectory with specified dynamic response and $r(n)$ is the bounded input to the reference model. If the tracking error is assumed to be small and the selected reference model is asymptotically stable, the motor speed at the (n+1) th sample can be forecasted from (3.1) as [13].

$$\omega^*_{ref}(n+1) = a_1\omega_r(n) + a_2\omega_r(n-1) + r(n) \tag{3.2}$$

Hence, with one sample of predicted speed and two samples of actual speed, an input sequence $\{\omega^*_{ref}(n+1), \omega_r(n), \omega_r(n-1)\}$ is formed and used as the input to the ANN 2, as shown in Fig. 3.1.

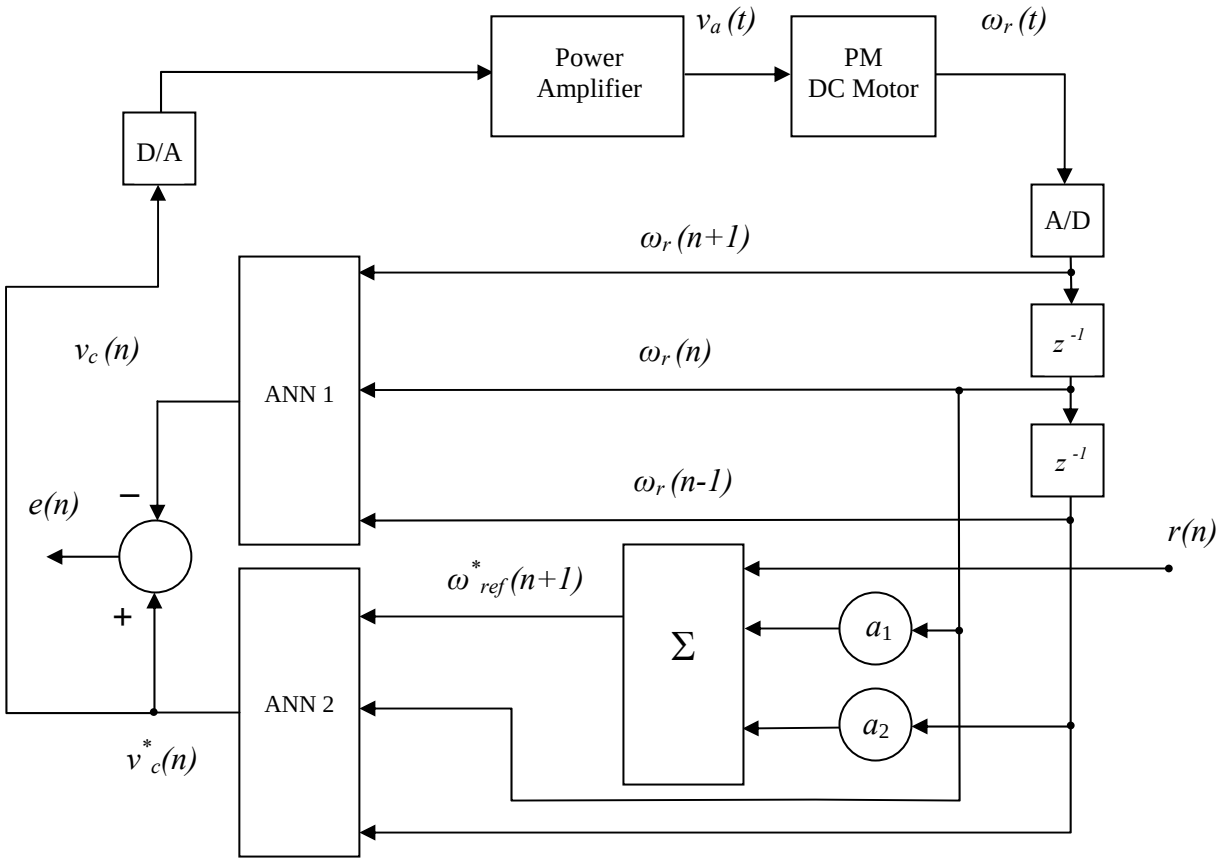

Figure 3.1 Block diagram for the ANN based Adaptive Controller.

The operation of the control system shown in Fig. 3.1 is as follows. During each sampling instance, the following steps are performed:

1. The set of inputs $\{\omega^*_{ref}(n+1), \omega_r(n), \omega_r(n-1)\}$ is applied to the controller to the controller ANN 2 to generate the control output $v^*_c(n)$.
2. $v^*_c(n)$ is then applied to the motor / driver system through the D/A converter.
3. The system waits until computer is interrupted, signifying the start of the next sampling period, say $(n+1)^{\text{th}}$ instance.
4. The new speed at the $(n+1)^{\text{th}}$ instance, $\omega_r(n+1)$ is measured.
5. The input set $\{\omega_r(n+1), \omega_r(n), \omega_r(n-1)\}$ is applied to ANN 1 to obtain the output $v_c(n)$.
6. The error e(n) = $e(n) = v^*_c(n) - v_c(n)$ is computed. $v^*_c(n)$ is the result from step 1 above while $v_c(n)$ is the result from step 5.
7. The error *e(n)* is back-propagated through ANN 2 , the weights of which are thus updated (or trained).
8. The same weight adjustments made to ANN 2 are also applied to ANN 1.
9. the procedure repeats from step1 with a new set of inputs $\{\omega_r(n+2), \omega_r(n+1), \omega_r(n)\}$ and so on.

As discussed in Chapter Two, the load model is given by (2.19), but this is not always the case in practical circumstances. This arise the need for on-line (adaptive) weights and biases updating of the ANN. However, the task of on-line training could be done without much difficulty and the system can be made more stable if an initial set of weights and biases is generated through the off-line trainings. A combination of off-line and on-line training has been used in this study. The initial set of weights and

biases were achieved through the off-line training. These weights and biases are updated only when an error limit between the actual output and the target of the ANN, exceeds a preset value.

3.2 Off-Line Training for Initial Set of Weights and Biases of the ANN

Data for off-line training can be obtained either by simulation or by experiment. If the motor parameters are available, then (2.23) can be used by randomly generating the inputs pattern of $\{\omega_r(n+1), \omega_r(n), \omega_r(n-1)\}$. The corresponding target can be generated by using these speed values and $K_1, K_2, K_3, K_4, K_5, K_6$ in the right-hand side of (2.23). Therefore off-line training data can be obtained by simulation using SIMULINK or any similar software in an open loop PM DC motor control scheme by considering the load as described in (2.19).

In the present work the experimental method was used. By using the experimental method we can get better results because in this case if there is any nonlinearity in the power amplifier, this also can be absorbed in to the ANN Structure for a better controller.

In the experiment, the PM DC motor was run in an open loop to follow a known arbitrary trajectory. This trajectory (see Fig.3.2) was generated in MATLAB. The speed of the DC motor and the supply voltage v_c to the amplifier was sampled at a rate of 1ms and recorded in the computer, which is interfaced to the DC motor. To record data the setup was done in Simulink (see Fig. B.1in Appendix B). From the recorded data, $\omega_r(n+1), \omega_r(n), \omega_r(n-1)$ and $v_c(n)$ were extracted to train the ANN. The ANN structure discussed under Section 2.5.2 was built in MATLAB and trained

to obtain the initial weights and the biases. These initial weights and biases of the ANN and the training curve of the ANN are given in Appendix B (see Fig. B.2 and Fig. B.3). To interface the computer and the DC motor the xPC target facility in the MATLAB® was used. The entire process of hardware and the software involved in the interface will be vigorously discussed in the next chapter.

3.3 On-Line Training for Weights and Biases and adaptive leaning of the ANN

The weights and biases of the two ANN's are updated at each instant using the back-propagation algorithm. The error function that is minimized is given by,

$$E(n) = \frac{1}{2} e^2(n) \tag{3.3}$$

$$e(n) = error = v_c^*(n) - v_c(n) \tag{3.4}$$

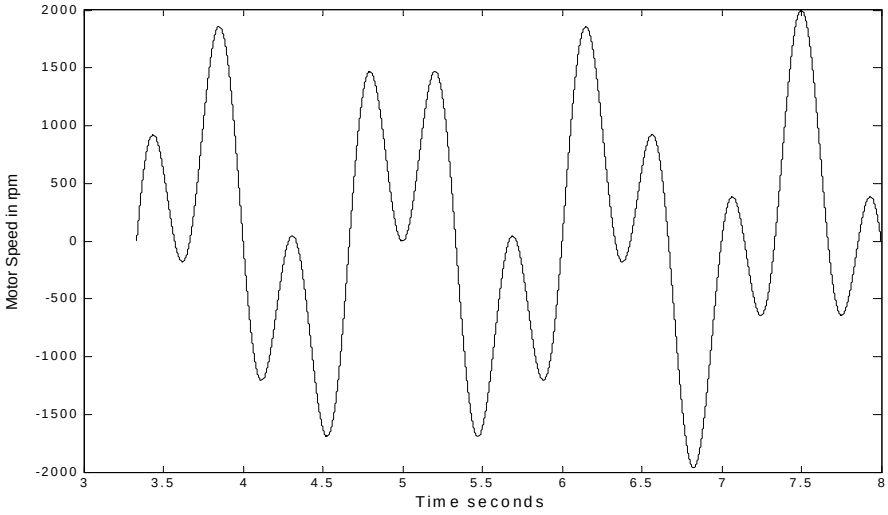

Figure 3.2 The trajectory generated for the DC motor to train the ANN.

Weights and biases of the output layer are updated according to the following expressions,

$$W_{jk}^{o}(n+1) = W_{jk}^{o}(n) + \eta\delta_{k}^{o}(n)O_{j}^{h}(n) \tag{3.5}$$

$$B_{k}^{o}(n+1) = B_{k}^{o}(n) + \eta\delta_{k}^{o}(n) \tag{3.6}$$

where η is the learning rate and $\delta_{k}^{0}(n)$ is the local gradient at the output layer, which can be expressed as,

$$\delta_{k}^{o}(n) = e(n)\frac{\delta e(n)}{\delta O_{k}^{o}(n)}S_{k}^{o}(n)(1-S_{k}^{o}(n))O_{j}^{h}(n) \tag{3.7}$$

Weights and biases of the hidden layer are updated according to the following expressions,

$$W_{ij}^{h}(n+1) = W_{ij}^{h}(n) + \eta\delta_{j}^{h}(n)I_{i}(n) \tag{3.8}$$

$$B_{j}^{h}(n+1) = B_{j}^{h}(n) + \eta\delta_{j}^{h}(n) \tag{3.9}$$

where $\delta_{j}^{h}(n)$ is the local gradient at the hidden layer, which can be expressed as,

$$\delta_{j}^{h}(n) = \delta_{k}^{o}(n)W_{jk}^{o}(n)[1-S_{j}^{h}(n)]^{2} \tag{3.10}$$

In real-time implementation of this setup the error is calculated at each instant and, when it exceeds a predetermined level, the weights and biases are updated. If the error is within a predetermined level, the previous set of weights and biases are retained to compute the control voltage $v_{c}(n)$.

Some of the main problems faced by high-performance motor drive applications are overshooting and response times. It has been observed that the learning rate η of the ANN is a key factor affecting for overshooting and response time. A more rapid learning rate causes overshooting on the speed, and a sluggish learning rate makes the response time too slow. Therefore, for on-line updating of

weights and biases of the ANN, an adaptive learning rate is introduced in our ANN controller. The initial learning rate of 0.0003 was obtained for the real-time implementation of the ANN controller base on the final value of the learning rate used in the off-line training.

We have considered the following facts when deriving the adaptive leaning rate η. If the difference between the reference speed and the actual speed is large, the learning rate is increased until the actual speed reaches the reference speed. Due to the faster learning rate, the actual speed may exceed the reference speed, resulting in overshooting. If an overshooting occurs, the learning rate is decreased. When the speed starts decreasing from the overshooting, the learning rate is again increased, so that the actual speed quickly reaches the reference speed. The detail of the adaptive learning rate is shown in the flowchart of Fig. 3.3.

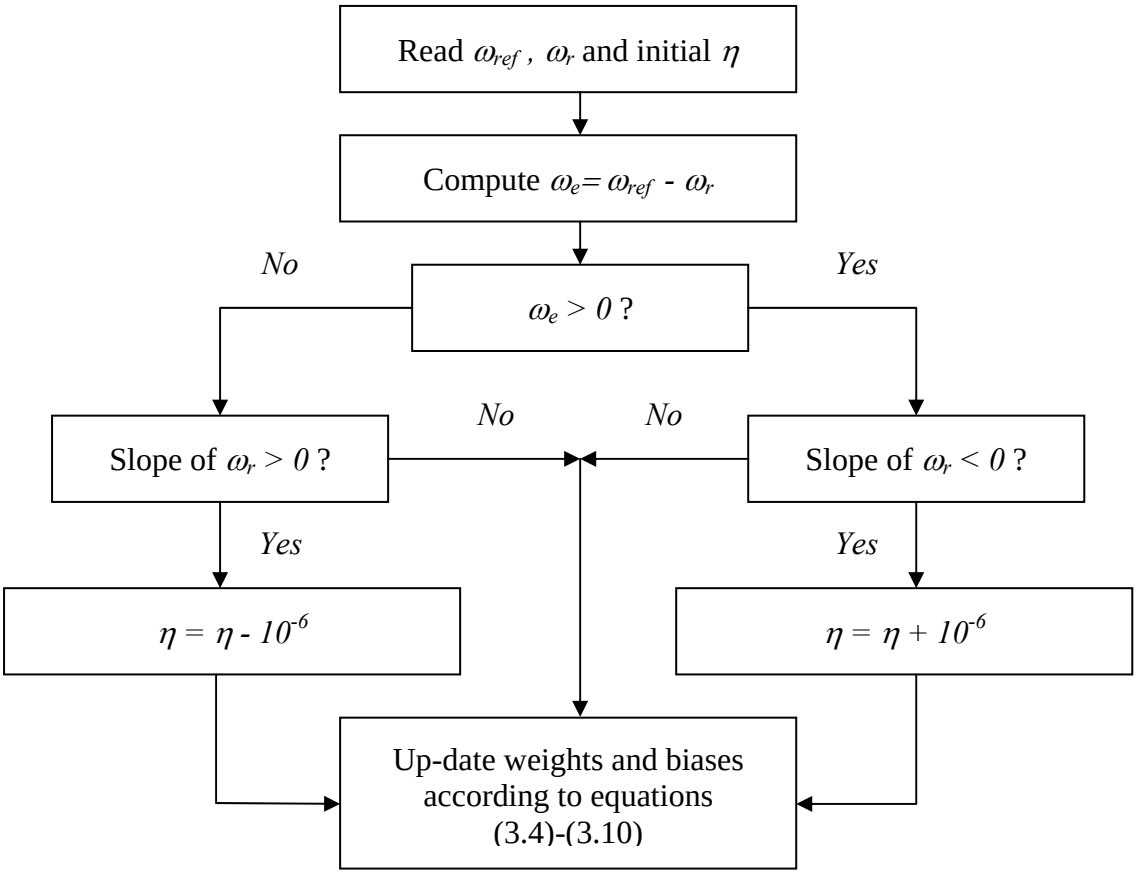

Figure 3.3 Flowchart for adaptive learning rate η.

3.4 Modified ANN Structure to Enhanced the Stability

Because the problem of overshoot and sluggish response mentioned in the previous section, the modified ANN structure shown in Fig. 3.4 was used. An extra feedback loop was added from the net input at the output neuron as shown.

It should be noted that the feedback connection has no connecting weight. In other words, the weight is fixed at one. Thus, the strength of this connection is not affected when connection weights and biases are updated during error back-propagation. The modified configuration of the ANN was found to provide stability on the performances of the motor controller. Extensive test were conducted and no instance of instability was experienced. Without this feedback loop, it was observed during the experiment that the motor speed and current were driven to saturation and the ANN based PM DC motor drive structure suddenly becomes unstable. We have over come this problem by providing the local feedback loop. The modified ANN structure with local feedback loop for the PM DC motor drive is shown in Fig. 3.4.

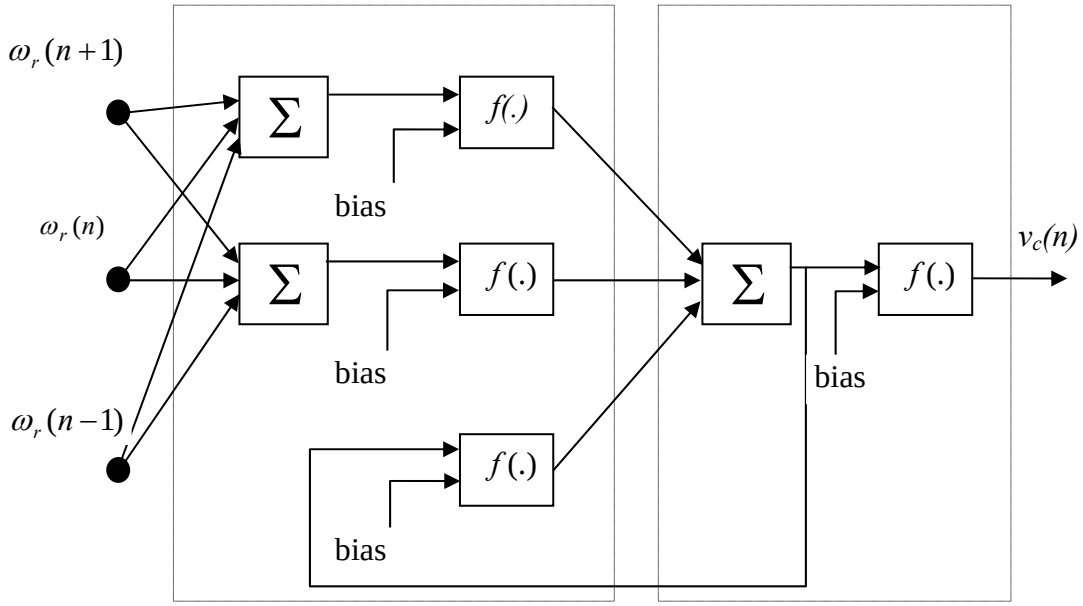

Figure 3.4 Modified ANN structure for PM DC motor drive.

3.5 Summary

The construction and training of the ANN based adaptive controller was presented in this chapter. First the architecture of the controller was presented and how the adaptations emulate to controller was discussed. Then, how the facility in the xPC target toolbox of the MATLAB was used for offline training and how the initial weights and biases of the ANN structure of the DC motor was obtained was discussed. More details on this issue will be discussed in Chapter four. Next, the on-line training of the ANN with adaptive learning rate was discussed. Finally, a better ANN structure, obtained by introducing a feedback loop within the ANN itself was presented which was found to improve the stability of the controller. The next chapter will present the real-time implementation of the experimental setup.

Chapter 4

REAL-TIME IMPLEMENTATION

This chapter describes the actual hardware and software implementation of the adaptive controller for the PM DC motor controller, which has been developed in Chapter Three and Chapter Four. Specifically, the techniques are implemented on a laboratory PM DC motor. The interfacing done by using MATLAB xPC target, which has a low cost real time implementation capability with hardware interface facility, is discussed in this chapter.

4.1 System Architecture

Since the advent of high level languages, the practice of developing software in a different environment to the environment in which it will eventually be used has become common. The development environment is referred to as the host, and the environment in which the software will be used is referred to as the target. This type of host-target real-time control system architecture shown in Fig. 4.1 was used in this work [18]. With a host computer running MATLAB, Simulink, Real-Time Workshop, xPC Target, and a C compiler as the development environment, real-time applications can be created. A desktop PC was used as the host running Windows. The model was built and real-time code was generated on the host computer. A second target PC is booted using a special boot disk that loads the xPC Target real-time kernel. After booting the target PC, one can then download the generated real-time application to it via the selected communication protocol, a TCP/IP network or a RS-232 serial link.

Here the TCP/IP communication is used because it is faster, providing data rates up to 10 Mbit/sec over any distance. Finally the real time execution is started from the host and the plant is controlled in real time at the target computer.

xPC Target supports a wide range of input/output (I/O) cards, which can communicate with the target computer. Simulink blocks represent the drivers. Interaction with the drivers is through these Simulink blocks and the parameter dialog boxes. An I/O card provides the interface between the target PC and the plant. The host and target computers and plant are connected as shown in Fig. 4.2.

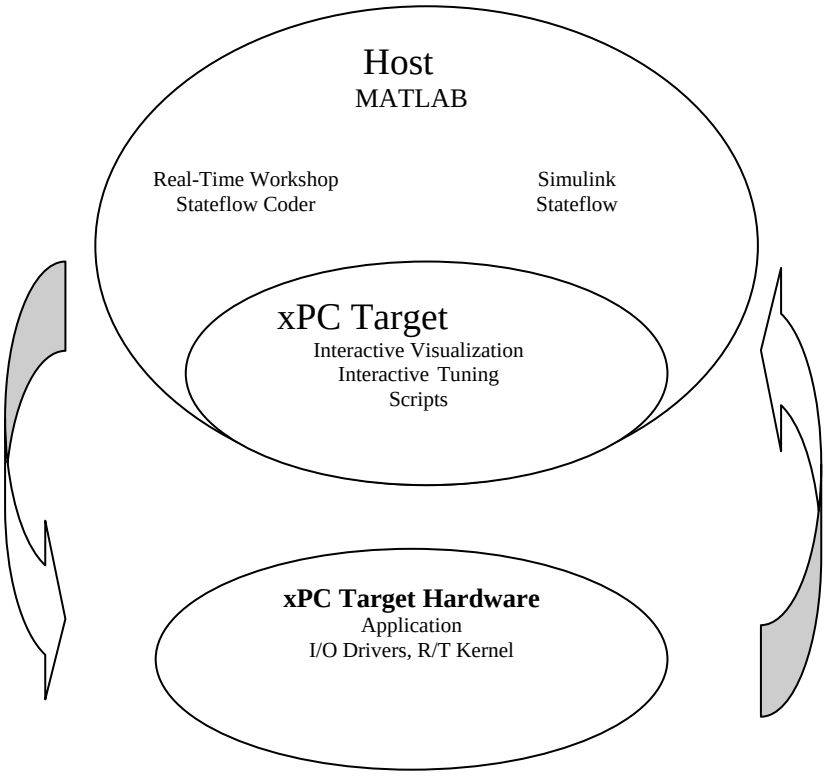

Figure 4.1 Host-target real-time control system architecture.

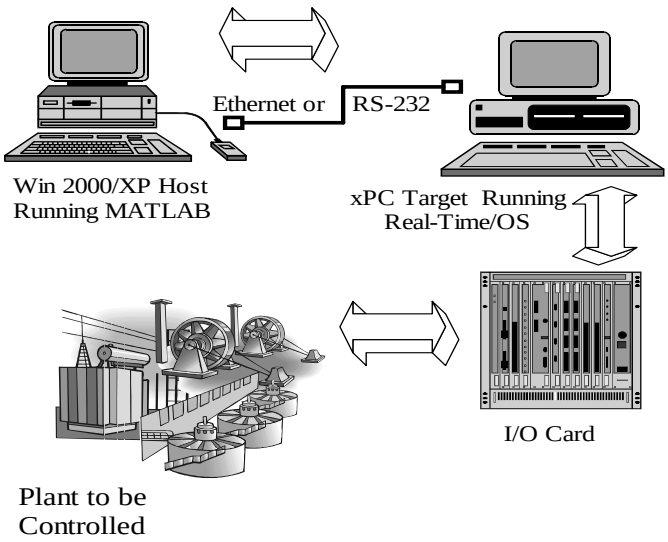

Figure 4.2 The host and target computer and Plant connection.

4.2 Hardware Interfacing

Two PC compatible computers, a host and a target, are required in the real time experimental setup. In this work, the host is a Pentium III 450 MHz with 128MB RAM, running Windows 2000 and other required software. The target is an AMD Athlon 1100MHz with 128 MB RAM and a custom made I/O card. The target PC is booted using a special boot disk that loads the xPC Target real-time kernel. The host and target computers communicate through Ethernet cards. These cards are connected through a direct connection using a cross-over Ethernet cable. The experimental laboratory setup including the DC motor is shown in Fig. 4.3.

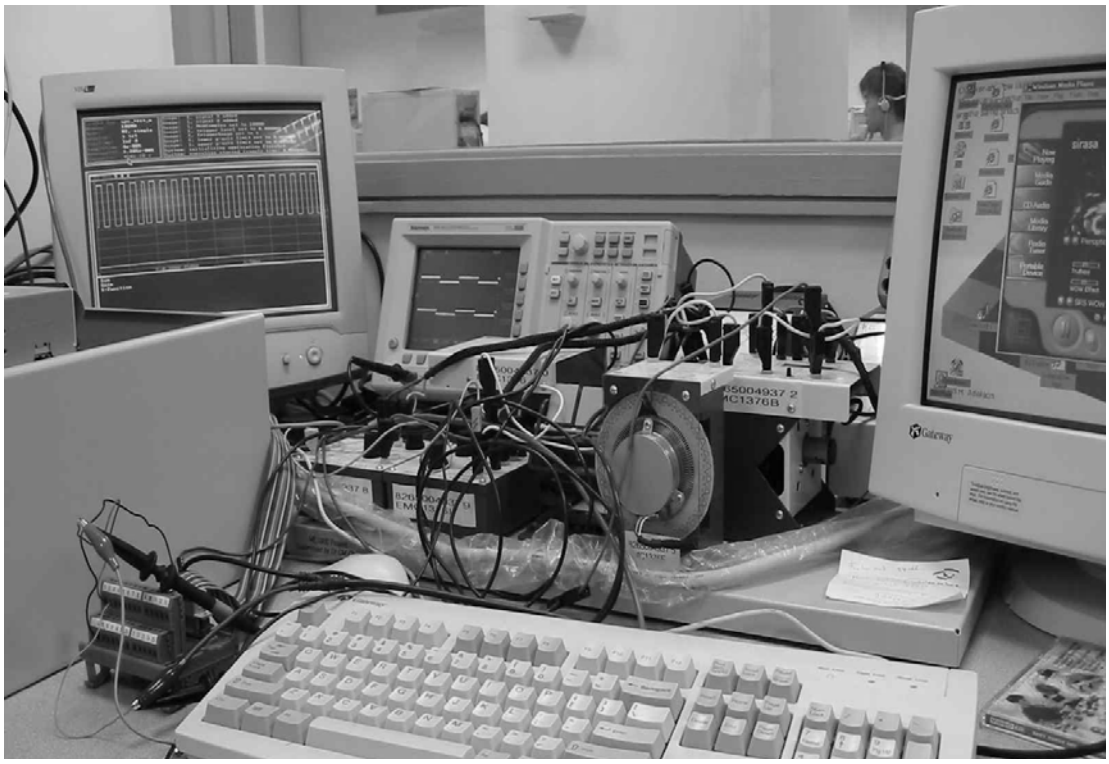

Figure 4.3 Experimental setup.

The I/O card, which consists of an analog-to-digital (A/D) and a digital-to-analog (D/A) converter, is the interface between the target PC and components of the DC motor drive, the velocity sensor and the current amplifier (which controls the current to the motor according to the input voltage). The velocity of the motor is measured through a DC tachogenerator. The input of the current amplifier (which controls the DC motor) is connected to the output from the D/A converter on the I/O card and the output of the tachogenerator is connected to the input of the A/D converter on the I/O card. The control voltage required to the current amplifier is computed inside the target computer.

4.3 Software Architecture

The adaptive controller is designed and developed using Simulink block diagram for xPC Target based prototyping environment, which is a real-time application facility in the MATLAB software. The real-time software compatible with MATLAB from MathWorks are the Real-Time Workshop (RTW), Real-Time Windows target, and xPC Target. The RTW and xPC Target requires a C compiler. The RTW and xPC produces codes for target applications directly from Simulink models and automatically builds and compiles using a C compiler. These are then down loaded to the target PC and can be run without the user having to write any low level codes.

A sample Simulink block diagram for this xPC Target based prototyping environment to control a DC motor is shown in Fig. 4.4. One of the advantages of this environment is the capability to substitute the S-Function Builder blocks with a Simulink block describing the dynamics of the plant/hardware to be controlled. Another advantage of this setup is the ability to modify the system parameter and the controller parameters on the host computer and observe their effect on the hardware in real-time.

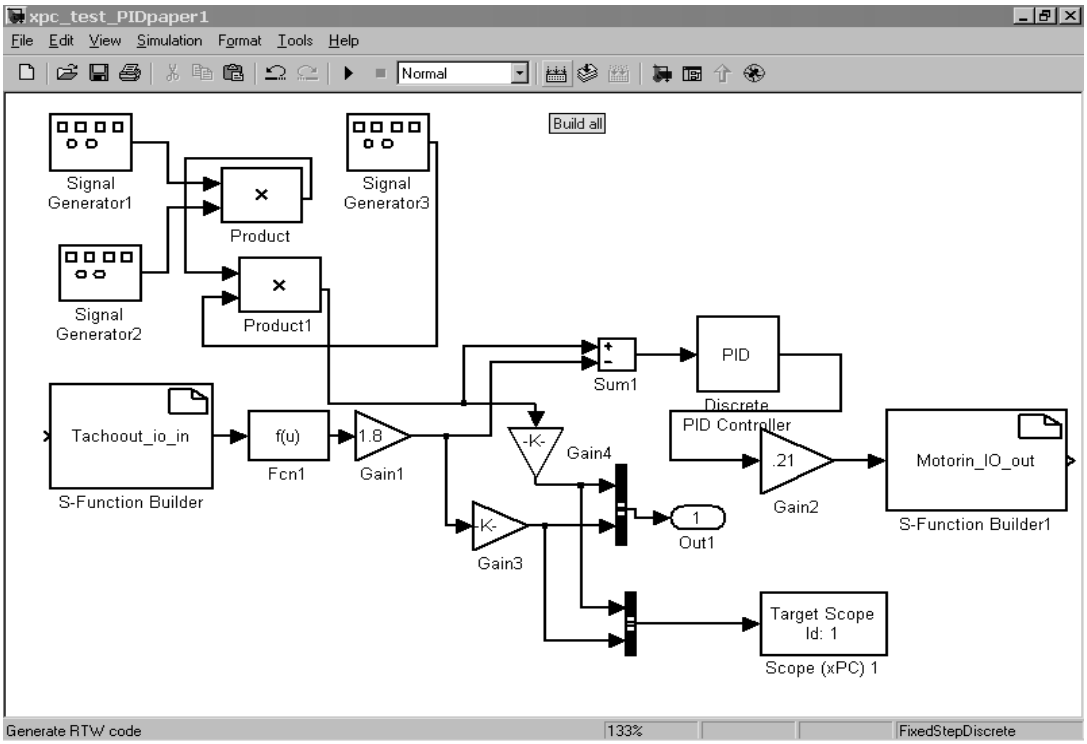

Figure 4.4 A sample Simulink block diagram for xPC Target based prototyping

4.4 Summary

The hardware and software development for implementing the adaptive controller for a DC motor with the objective of real-time performance evaluation was described here. The chapter mainly discusses how a real-time control system for the PM DC motor controller is developed using hardware components (standard PC and tailor made I/O card) and readily available software tools (MATLAB, Simulink, xPC target, RTW and Watcom C/C^{++} compiler). The main advantage of the discussed prototyping real-time environment is that it can be used for development of control algorithms for the adaptive controller without the need to write low –level software codes. The next chapter will present the results of experimental investigations and performance evaluation of the ANN based adaptive controller.

Chapter 5

EXPERIMENTAL RESULTS AND OBSERVATIONS

This chapter presents an experimental investigation of the ANN based adaptive controller for the PM DC motor discussed in the previous chapters. The results of representative experimental cases obtained from the adaptive control of the DC motor are presented and analyzed. For comparison purpose some results were taken by using PI and PID controllers. The results from the PI and PID controllers were compared with an ANN based adaptive controller. This chapter is concluded with a discussion of the robustness, efficiency and reliability of the ANN based DC motor controller.

5.1 Verify the validity of ANN motor model

After off-line training has been done to the ANN model for the motor drive using the speed trajectory shown in Fig. 3.2 as discussed in the Section 3.2 some testing were carried out to verify the validity of the ANN motor model. To verify the validity, a setup was built in the Simulink (see Fig. C.1 in the Appendix C). In this setup the motor and the ANN model both were run at the same time in an open loop to follow a known single arbitrary trajectory. This trajectory is different from the trajectory used to train the ANN structure. Both the response of the motor and that of the ANN are shown in Fig. 5.1. The error between the two trajectories are shown in Fig. 5.2

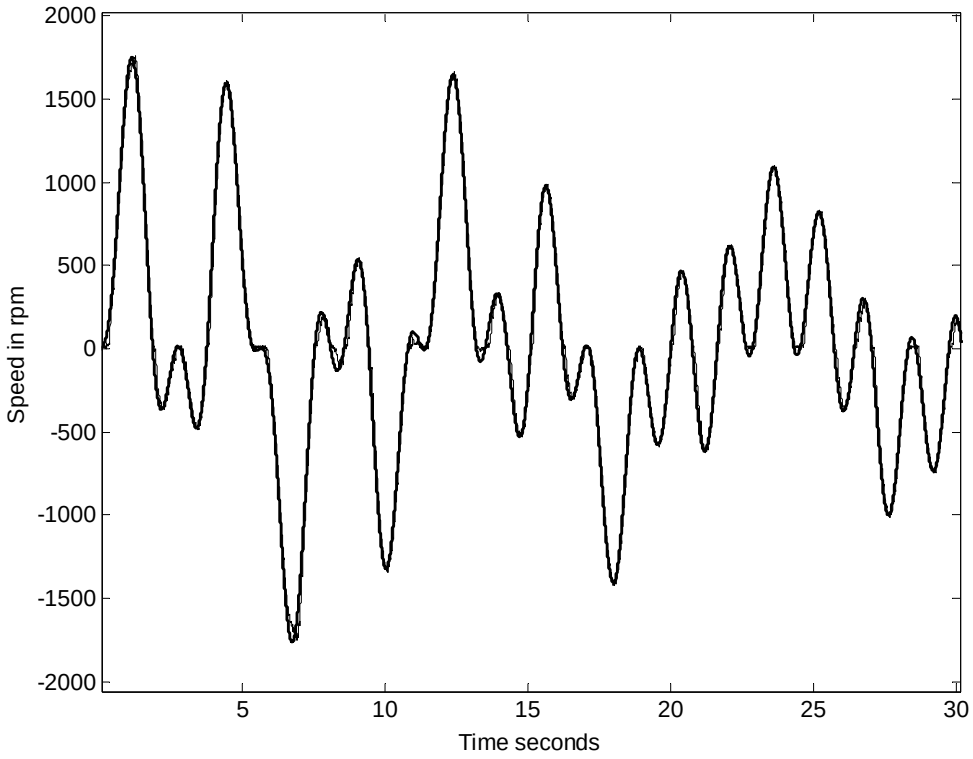

Figure 5.1 Out put trajectory of the motor and the ANN model solid line represent the motor output and dotted line represents the ANN model output.

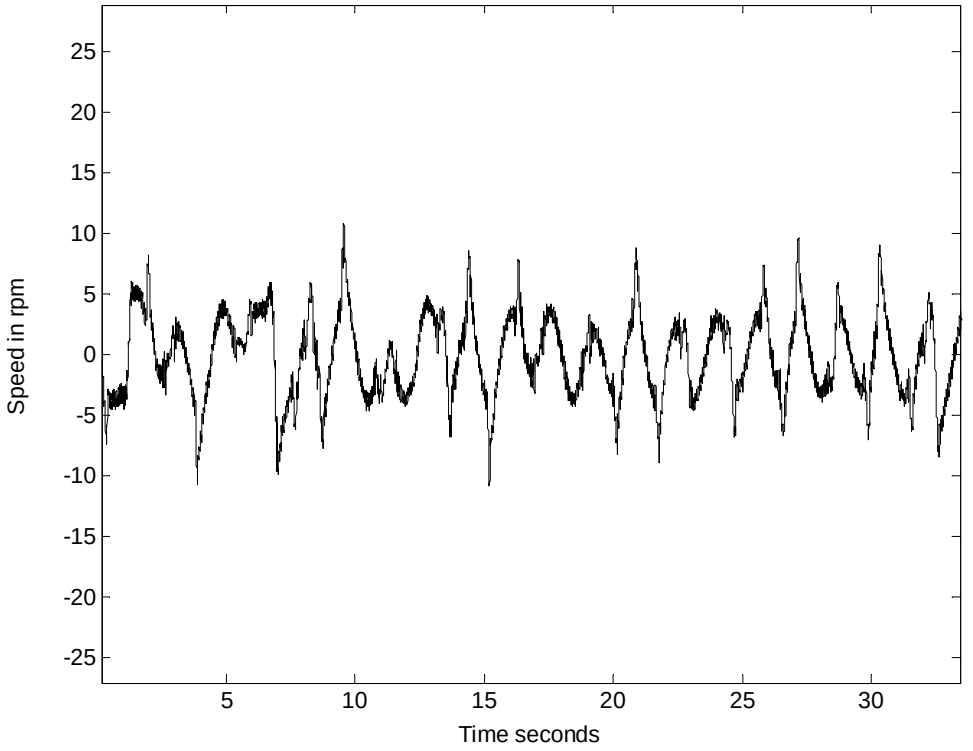

Figure 5.2 Error between the two out put trajectories.

When comparing the two trajectories, the output speed trajectory of the motor and the out put speed trajectory of the ANN motor drive model as shown in the Fig. 5.1, it is observed that the two trajectories are very close to each other. This indicates that the ANN motor drive model that was created to mimic the motor model strongly represents the actual motor. Further more this can be verified by observing the error between the two trajectories. This error is represented in Fig. 5.2. While the motor was run to more than 1500 rpm, the error in speed is less than about 10rpm.

5.2 ANN based adaptive controller

After carrying out the tests and verifying the validity of the ANN motor model, the ANN based adaptive controller was built (see Fig. D.1 in Appendix D) in the Simulink environment by using the verified ANN motor model. The structure of the ANN based adaptive controller was built as described in the Section 3.1 of Chapter Three. After that the ANN based adaptive controller was coded and downloaded to the target computer in the xPC host-target enviremnet. Then various tests were carried out and some were compared with a PI and PID controller to find out the performances and the adaptability of the ANN based adaptive controller. The PI and PID controllers were designed for the critically damped response of the speed with the best estimation of the motor parameters. When the experiment was carried out the constants a_1 and a_2 were chosen as 0.6 and 0.2 respectively for a critically damped response.

5.2.1 Responses for varying reference speed steps with full load

Step responses were performed to obtain the speed responses for some reference speed steps with full load. Figures. 5.3 - 5.5 show the speed responses of the ANN based adaptive controller, PID controller and PI controller respectively. In these figures, the dotted curves represent the reference input, $\omega^*_{ref}(n)$, where the solid curves represent the responses.

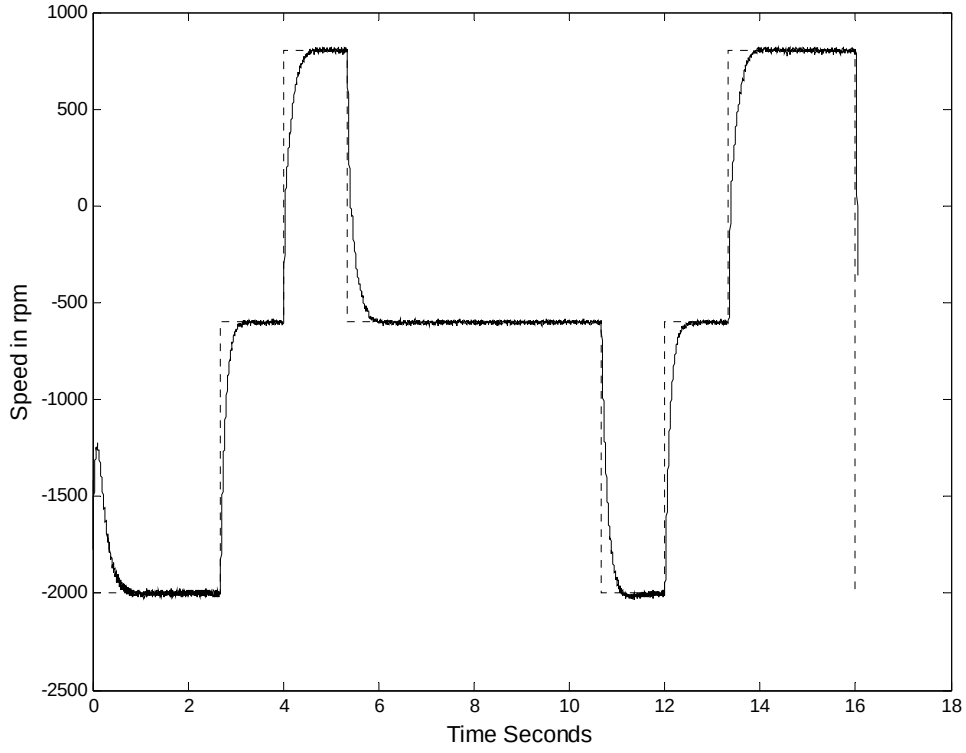

Figure 5.3 Experimental result of the ANN based controller with changes in reference speed.

It is observed from the figures that the ANN based adaptive controller has performed much better than the other two controllers. Both the PID and PI controllers had the problem of overshooting when there was a change of reference speed. In both the controllers the constants were chosen to provide the critically damped speed response based on the available design parameters.

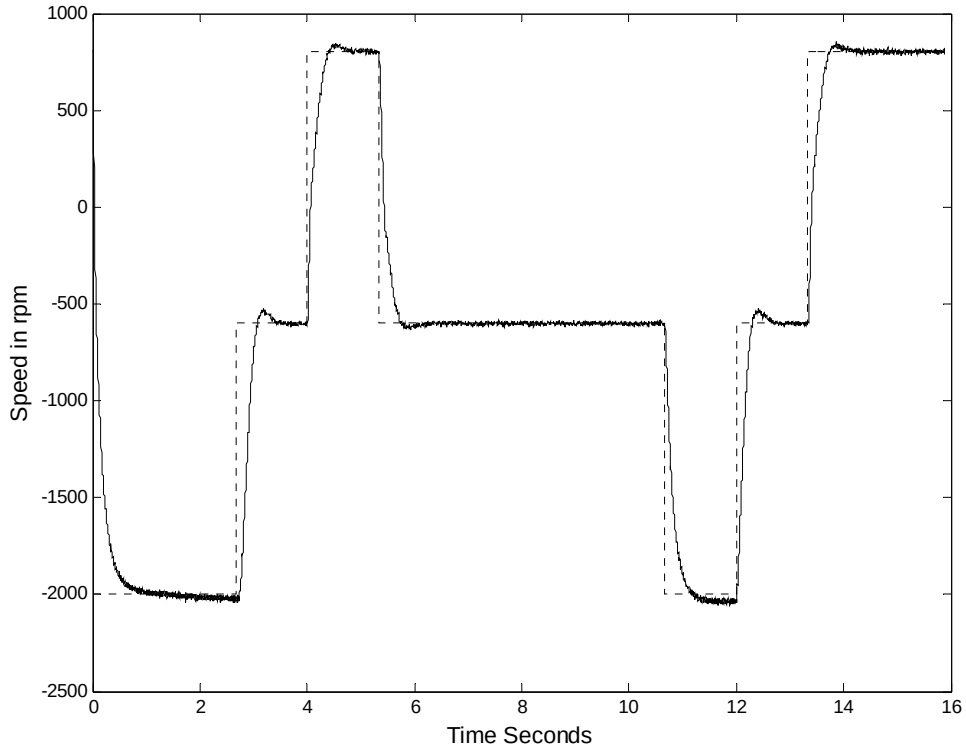

Figure 5.4 Experimental result of the PID controller with changes in reference speed.

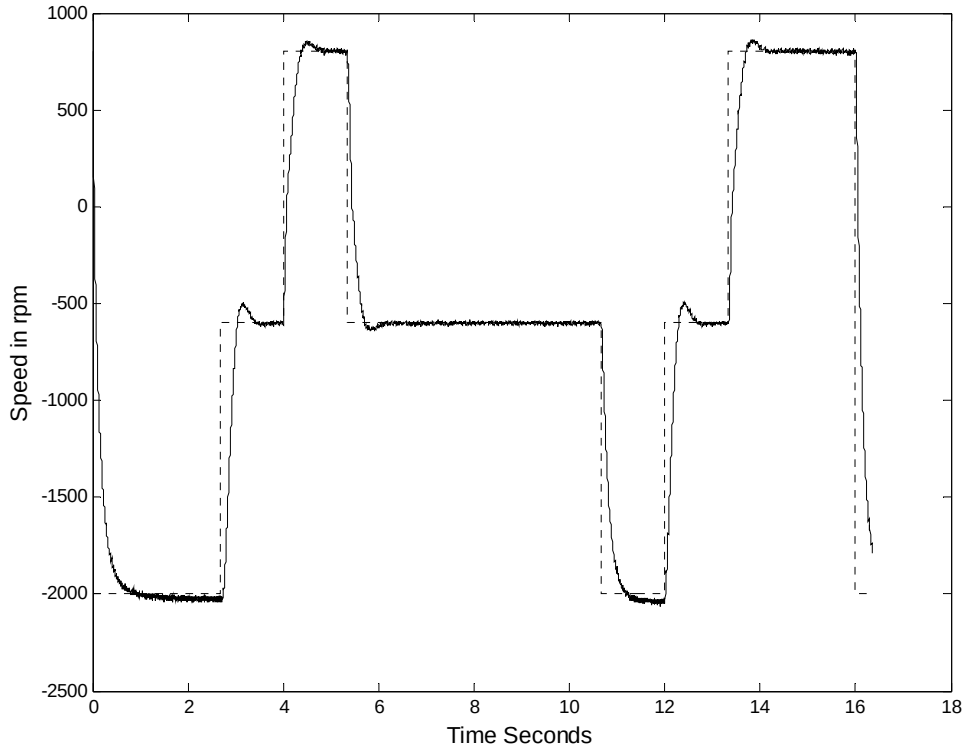

Figure 5.5 Experimental result of the PI controller with changes in reference speed.

The PID and PI controllers may be redesigned for more overdamped condition to reduce the overshooting. In such cases it was observed that the rise time became more sluggish . It can be seen too that the steady-state error performance of the ANN was also significantly better than for both the PI and PID controllers. The speed response of the ANN based adaptive controller was more robust against the change in the operating condition of step changes in the reference speed, because of the on-line tuning of the weights and biases in the ANN model.

5.2.2 Responses for a speed trajectory

Here some tests were performed to obtain the actual and specified speed trajectories for a sinusoidal type input. Figures 5.6 - 5.8 show the actual and specified speed trajectories of the ANN based adaptive controller, PID controller and PI controller respectively.

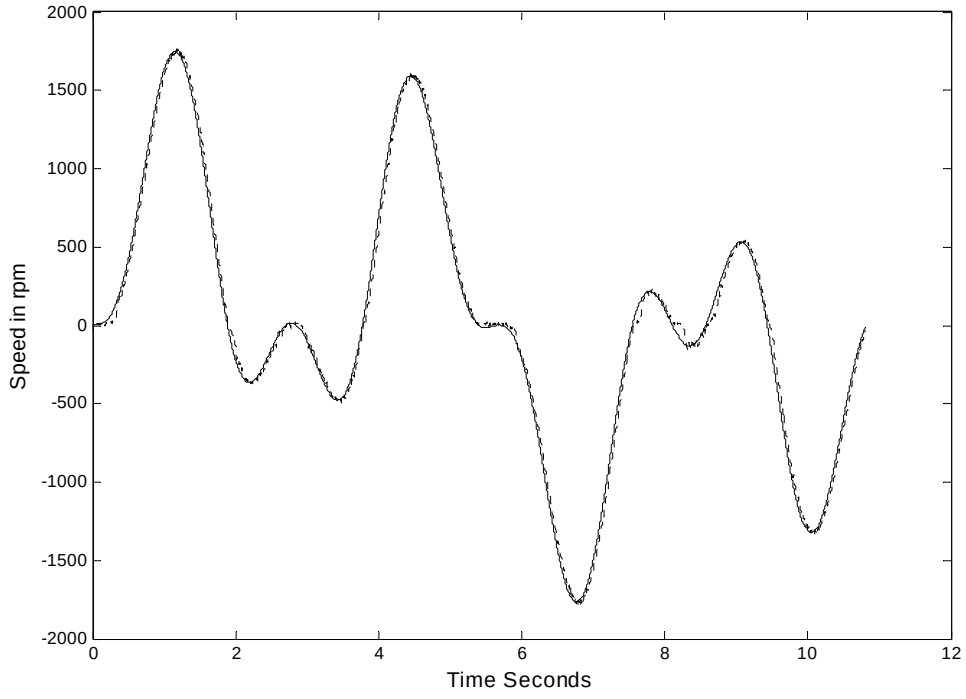

Figure 5.6 Response of the ANN based controller with changes in sinusoidal type reference speed track.

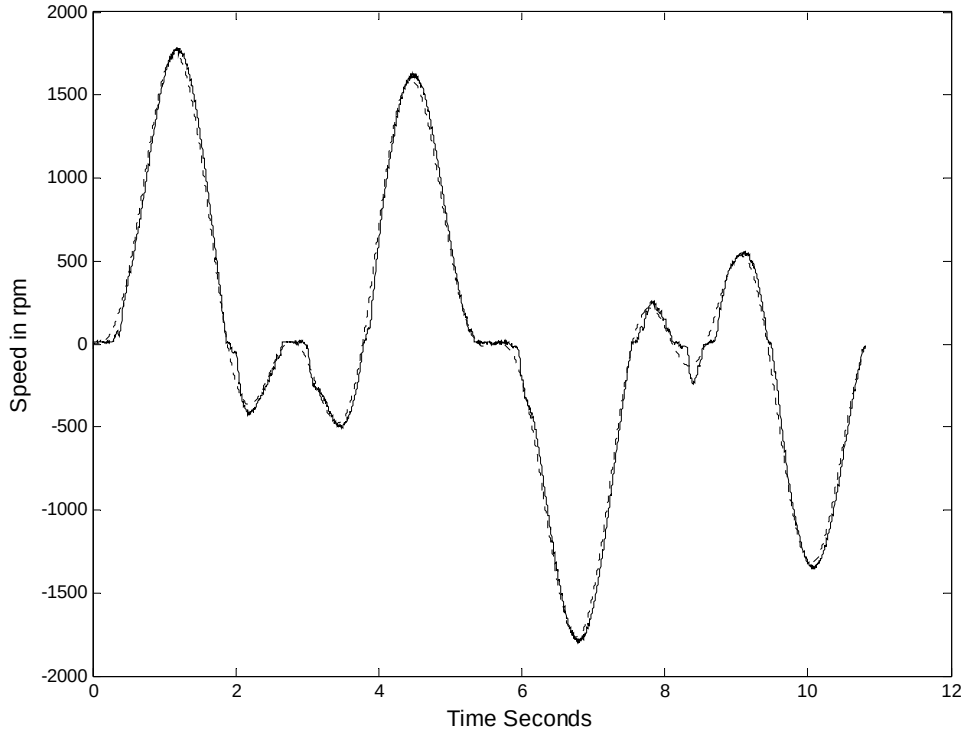

Figure 5.7 Response of the PID controller with changes in sinusoidal reference speed track.

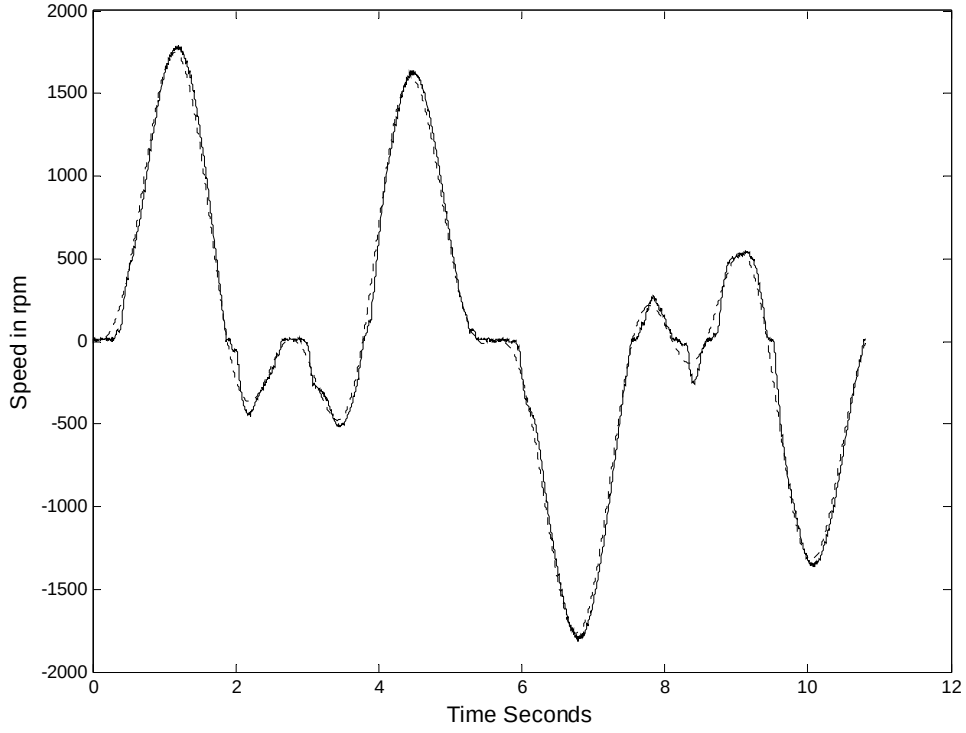

Figure 5.8 Response of the PI controller with changes in sinusoidal reference speed track.

5.2.3 Tracking performance with noise added

Another sets of tests were performed to study the effect of measurement noise on the performance. White noise was injected into the system by adding this to the values of $\omega(n+1)$ which was measured with the A/D converter. The following figures Fig. 5.9, Fig. 5.10 and Fig. 5.11 show respectively the actual and specified speed trajectories of the ANN based adaptive controller, PID controller and PI controller after noise was introduced to the systems.

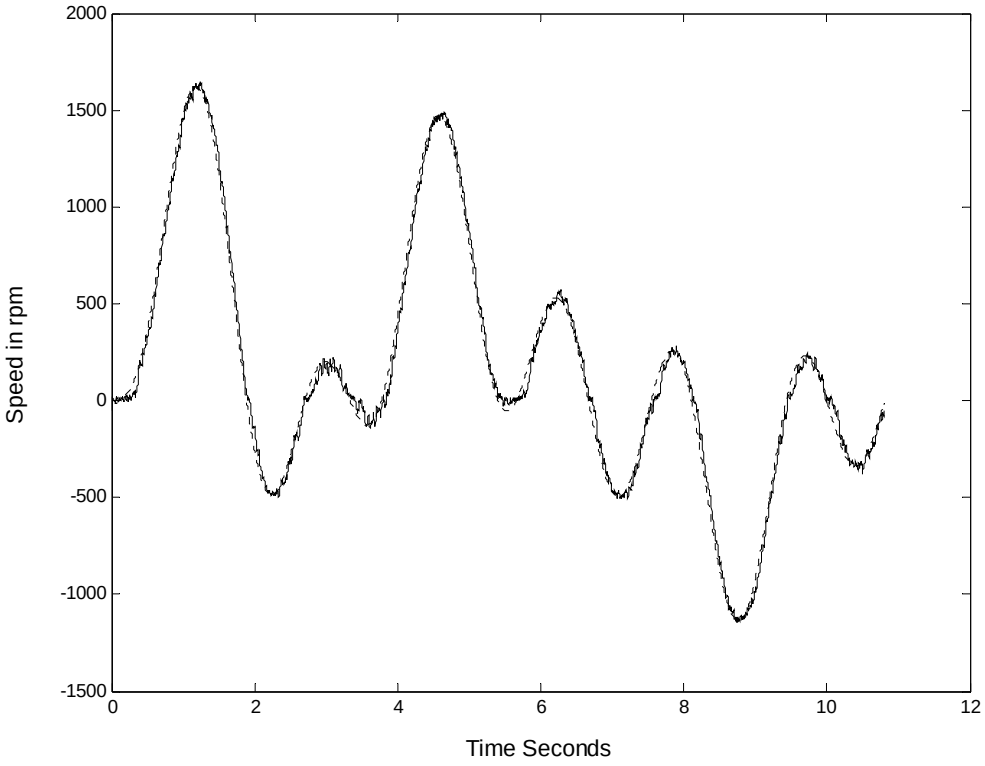

Figure 5.9 Tracking performance of the ANN based controller with noise.

As can be clearly seen from the figures, the ANN based controller displays high tracking accuracy at all speeds as compared with both the PID and PI controllers when noise was introduced to the systems. The maximum tracing error is 40 rpm or 2.5 % of the maximum Trajectory speed for the system under ANN control. These

compares with 95 rpm (6 %) and 130 rpm (8 %) for the PID and PI control respectively. This displays the capability of noise rejection of the ANN based controller.

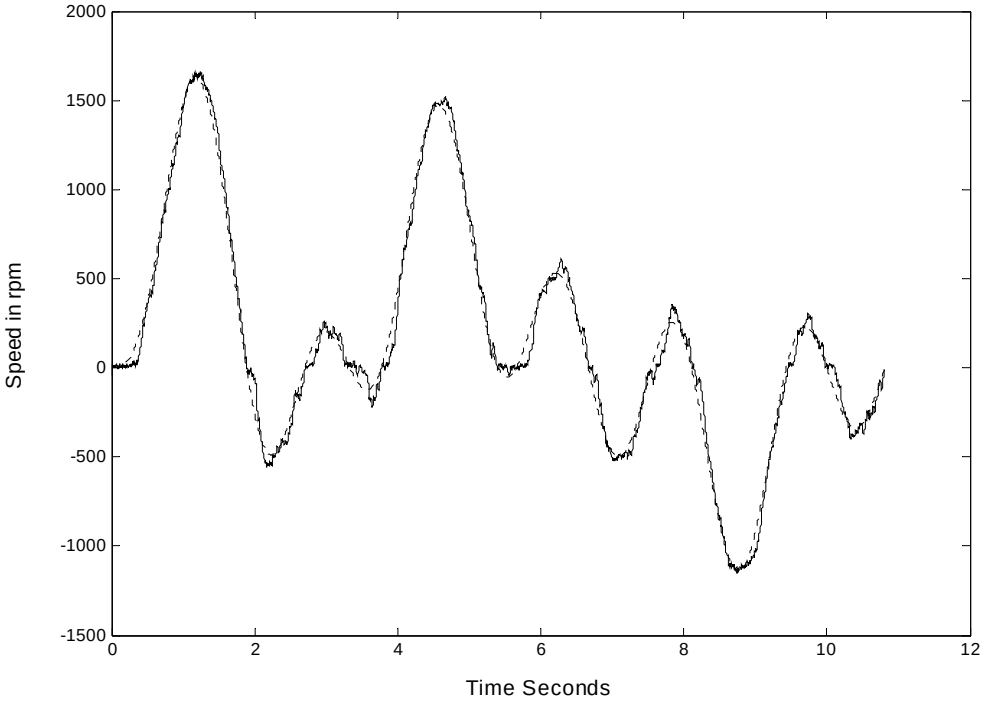

Figure 5.10 Tracking performance of the PID controller with noise.

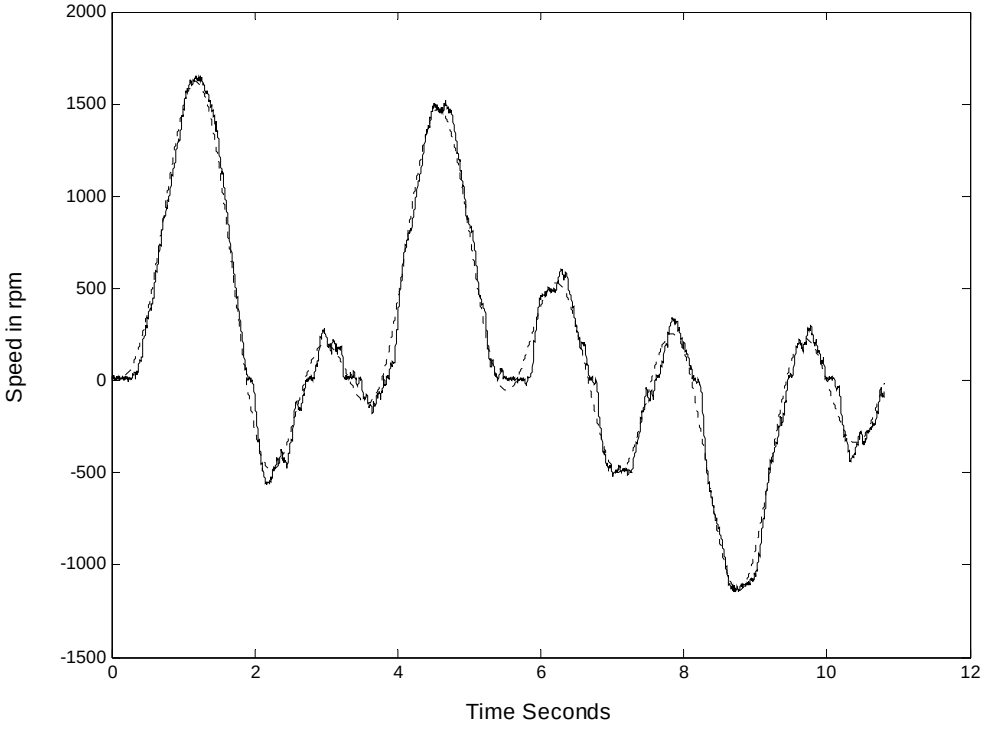

Figure 5.11 Tracking performance of the PI controller with noise.

5.2.4 Responses when the rated load is applied suddenly

These tests were performed to study the effect when the rated load is applied suddenly to the systems. Figures Fig. 5.12, Fig. 5.13, and Fig. 5.14 show speed responses of the ANN based adaptive controller, PID controller and PI controller respectively when the motor was running at no-load condition and, after some time, the rated load (exactly the same load is applied to each controller) is applied suddenly.

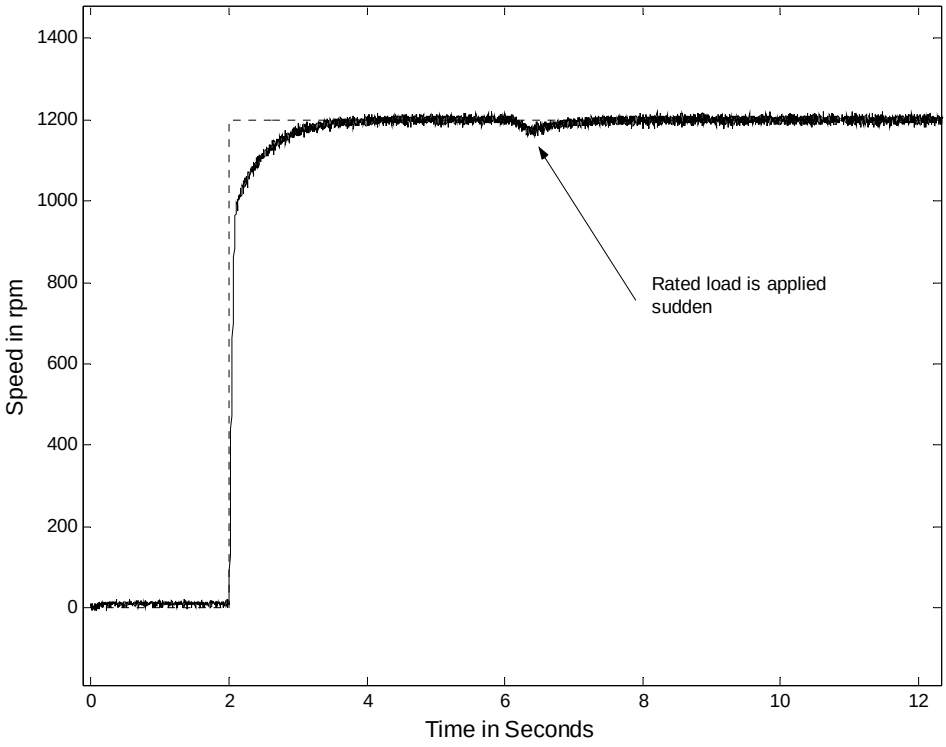

Figure 5.12 Speed of the ANN based controller with step change in the load.

From the figures, it can bee seen that the response of the ANN based controller system is superior to the other two controllers. The ANN adjusts its weights and biases to this changing condition of a sudden load impact and provides appropriate control voltage, so that the drive system responds according to the reference speed.

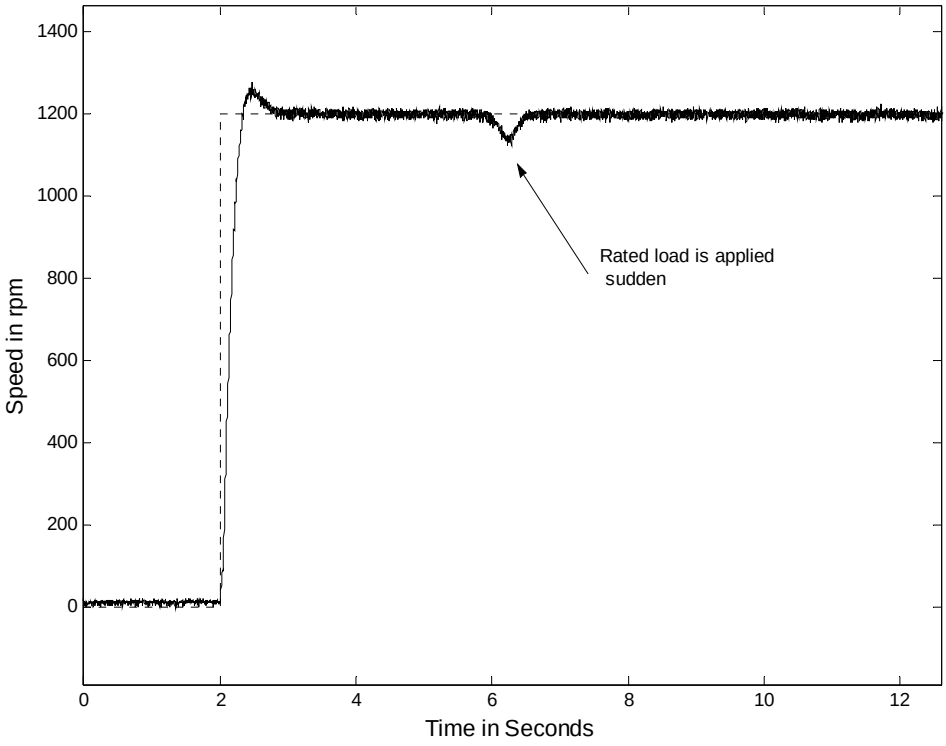

Figure 5.13 Speed of the PID controller with step change in the load.

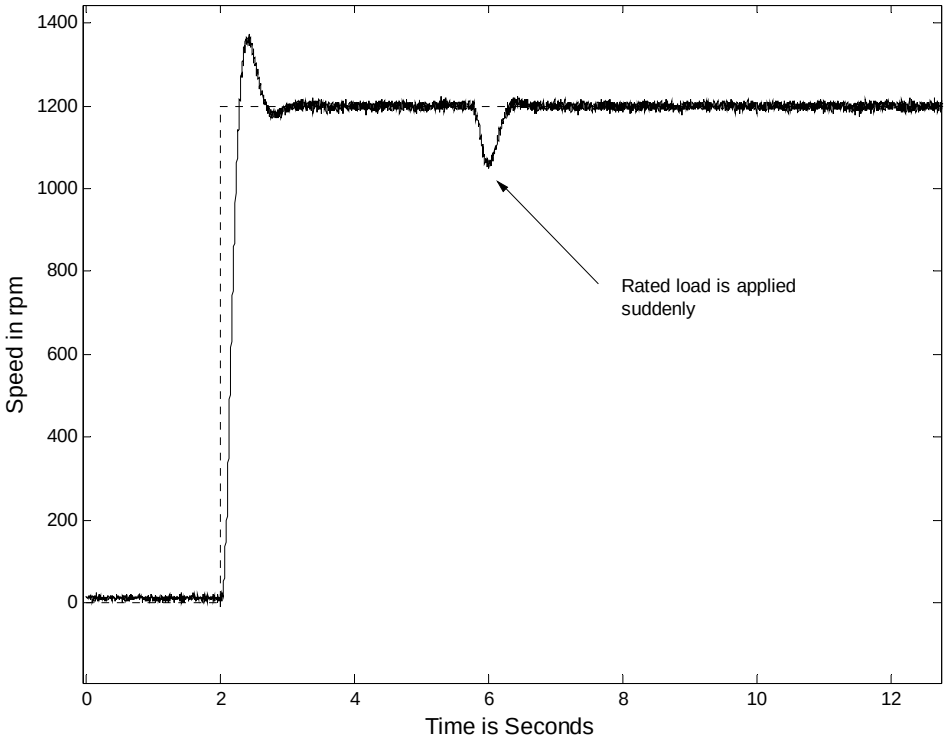

Figure 5.14 Speed of the PI controller with step change in the load.

5.3 Discussion

In the present work, a practical design development, implementation and experimentation have been carried out for an ANN based adaptive controller. At this stage the advantages of the adaptive controller can be identified. From all the tests conducted it can be seen that the ANN based controller is superior to both the PID and PI controllers. As expected, due to the on-line weights and biases updating of the ANNs in the ANN based controller with adaptive learning rates, the proposed controller becomes insensitive to noise and sudden load variations and thus, tracks the reference speed quite accurately. Also it can be observed that the speed overshooting of the ANN based controller is significantly lower than the other controllers. This critically damped speed response has been achieved using the adaptive leaning rate feature in the ANN based controller. Furthermore, the added feedback loop in the ANN inner structure in the ANN based controller has, from the extensive tests conducted, eliminated the problem of instability experienced by ANN controller without this loop. Since ANN can effectively represent nonlinear aspects, the ANN based adaptive controller also has the potential of better compensating for the non-linearity of the motor that the other two controllers does not have.

5.4 Summary

Experiments were conducted in real-time and the results were gathered from a host-target prototype system. In this experiment a laboratory DC motor, which has loading facility, was used. To control the motor an ANN based adaptive, PID and PI

controllers were implemented using the system. Several sets of experiments and studies for evaluation were carried out to compare the ANN based controller with the other two PID and PI controllers. Firstly the DC motor was run in open loop to get some actual data to train the ANN motor model. After that the motor speed was controlled using the three controllers under similar conditions and the results were obtained. The next chapter presents the conclusion and the recommendations of the project.

Chapter 6

CONCLUSION AND RECOMMENDATIONS

6.1 Primary Contributions

This thesis investigated the effectiveness of enhancing adaptive control using them in practical systems. Artificial Neural Network was used as a trainable non-linear mapping system. The speed of a permanent magnet direct current motor was controlled using the proposed ANN based adaptive controller. The details of development of the proposed controller were presented, including all analytical derivations. Programming and implementation details including hardware interfacing were given as well, for both the computer setup and the physical experimentation. The results obtained in real-time using the proposed ANN based adaptive controller were used to evaluate the performance of the controller. Then the results of the comparison of the proposed controller with the traditional PID and PI controllers were also presented in this study.

It is obvious in the real world that most of the plants show non-linearity and it is a fact that most of the controllers such as simple PID and PI are unable to achieve high accuracy due to this phenomenon. During the experimentation and after observing the results it has been proved that the proposed ANN based controller has a good ability to control the speed of the PM dc motor, which shows the non-linearity behavior. Experimental results verify that this ANN based adaptive controller performed much better than both the PID and PI controllers and was able to reduce

the tracing error to less that 3%. We can come to a conclusion that the proposed artificial neural network based adaptive controller is clearly superior, particularly in the case of non-linearities, parameter variations and load disturbances. The on-line weights and biases updating feature of the ANN can compensate for both parameter changes and disturbances during operation. The uses of the adaptive learning rate in the proposed controller reduce the possibility of overshooting particularly during the transient conditions. The feedback provision in the modified ANN motor structure also enhances the stability of the system. The proposed ANN based speed controller of the PM dc motor was found to be robust, efficient and easy to put into practice.

6.2 Further Studies

While the research reported in this thesis shows that an ANN based adaptive controller performance is superior it still lacks with some limitations, which provides room for improvement. Such possible improvements are indicated below, as possible directions for further work.

In the present work the number of hidden layers and the number of neurons in the hidden layer are chosen by trial and error, bearing in mind that the smaller the number, the better it is in terms of both memory and time taken to implement the ANN. Further research can be done to find the optimum number of hidden layers and number of neurons in the hidden layer.

This research did not make attempt to compare the stability and performance of the ANN based adaptive controller over conventional adaptive control techniques, such as model reference adaptive control, sliding mode control and variable structure control. One can try to do further work in this direction.

Bibliography

[1] Astrom, K. J. and B. Wittenmark, *Adaptive Control*, Addison-Wesley, Reading, MA, 1995.

[2] El-khouly, F. M., A. S. Abdel-Gaffer, A. A. Mohammed, and A. M. Sharaf, "Artificial intelligent speed control strategies for permanent magnet dc motor drives," in *Proc. IEEE-IAS Annu. Meeting,* 1994, vol. 1, pp. 379–384.

[3] Fukuda, T.; Shibata, T.**,** **"**Theory and applications of neural networks for industrial control systems" Industrial Electronics, IEEE Transactions on , Volume: 39 Issue: 6 , Dec 1992 pp 472 –489

[4] Hoque, M.A., M.R. Zaman, and M.A. Rahman, "Artificial neural network based controller for permanent magnet dc motor drives," in *Proc. IEEE-IAS Annu. Meeting,* 1995, vol. 2, pp. 1775–1780.

[5] Hoque, M.A., M. R. Zaman, and M. A. Rahman, "Artificial neural network based permanent magnet dc motor drives," in *Proc. IEEE-IAS Annu. Meeting,* 1995, vol. 1, pp. 98–103

[6] Levin, A.U.; Narendra, K.S.; "Control of nonlinear dynamical systems using neural networks: controllability and stabilization" Neural Networks, IEEE Transactions on , Volume: 4 Issue: 2 , Mar 1993 pp 192 –206

[7] MATLAB *Inc. http:///www.mathwork.com*

[8] Narendra, K.S.; Parthasarathy, K "Identification and control of dynamical systems using neural networks" Neural Networks, IEEE Transactions on, Volume: 1 Issue: 1, Mar 1990 pp 4 –27

[9] Narendra, K.S**.,** **"**Neural networks for control theory and practice" Proceedings of the IEEE , Volume: 84 Issue: 10 , Oct 1996 pp 1385 –1406

[10] Narendra, K.S. and S. Mukhopadhyay, "Adaptive control using neural networks and approximate models" Neural Networks, IEEE Transaction on Volume: 8 Issue: 3 , May 1997.

[11] Narendra, K.S., " Neural networks for real-time control" Proceedings of the 36th Conference on Decision & Control, San Diego, California USA, December 1997.

[12] Rubaai, A. and R. Kotaru, "Online identification and control of a DC motor using learning adaptation of neural networks" Industry Applications, IEEE Transactions on Volume: 36 Issue: 3, May-June 2000.

[13] Weerasooriya, S. and M.A. El-Sharkawi, “Identification and control of a DC motor using back-propagation neural networks" Energy Conversion, IEEE Transaction on Volume: 6 Issue: 4, Dec. 1991.

[14] Weerasooriya, S. and M.A. El-Sharkawi, “Laboratory implementation of a neural network trajectory controller for a DC motor” Energy Conversion, IEEE Transaction on Volume: 8 Issue: 1 , March 1993.

[15] Isermann, R., K.-H. Lachmann, and D. Matko, *Adaptive Control Systems*, Prentice-Hall Inc., Englewood Cliffs, NJ, pp.245-255, 1992.

[16] Ogata, K., *Discrete-Time Control Systems*, Prentice-Hall Inc., Englewood Cliffs, NJ, pp.203, 315-318, 1987.

[17] Haykin, H, *Neural Networks: A Comprehensive Foundation.* Piscataway,NJ: IEEE Press, 1994.

[18] De Mel, W.R. and A. N. Poo, “ Real-Time Control using xPC-Target in MATLAB” International Symposium on Dynamics and Control, Hanoi, Vietnam, September, 2003. (Submitted)

Appendix A

$$K_1 = \frac{2L_aJ + T_s(R_aJ + L_aB) - T_s^2(R_aB + K_EK_T)}{L_aJ + T_s(R_aJ + L_aB)} \quad \text{(A.1)}$$

$$K_2 = -\frac{L_aJ}{L_aJ + T_s(R_aJ + L_aB)} \quad \text{(A.2)}$$

$$K_3 = -\frac{T_s(\nu L_a + R_aT_s)}{L_aJ + T_s(R_aJ + L_aB)} \quad \text{(A.3)}$$

$$K_4 = \frac{T_s\nu}{L_aJ + T_s(R_aJ + L_aB)} \quad \text{(A.4)}$$

$$K_5 = \frac{K_TT_s^2}{L_aJ + T_s(R_aJ + L_aB)} \quad \text{(A.5)}$$

$$K_6 = -\frac{T_FR_aT_s^2}{L_aJ + T_s(R_aJ + L_aB)} \quad \text{(A.6)}$$

Appendix B

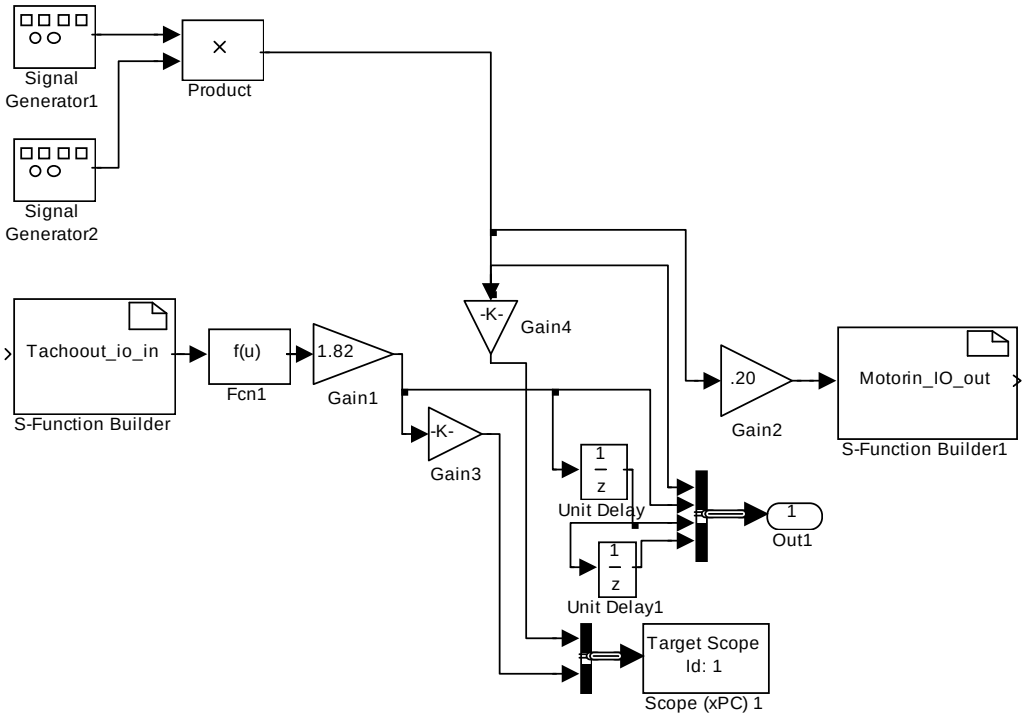

Fig B.1 Simulation schematic diagram for the dc motor in open loop to obtain the experimental data to train the ANN for initial weights and biases

Weights to hidden layer		
-5.6857	-2.9143	0.32942
1.9961	-0.0090955	-1.9238
38.9991	-4.1948	-27.1007
Bias to hidden layer		
-17.6709	-0.91774	13.7385
Weights to Output layer		
8.8373	54.3715	19.2375
Bias to Output layer		
29.0305		

Fig B.2 Initial Weights and Biases of ANN

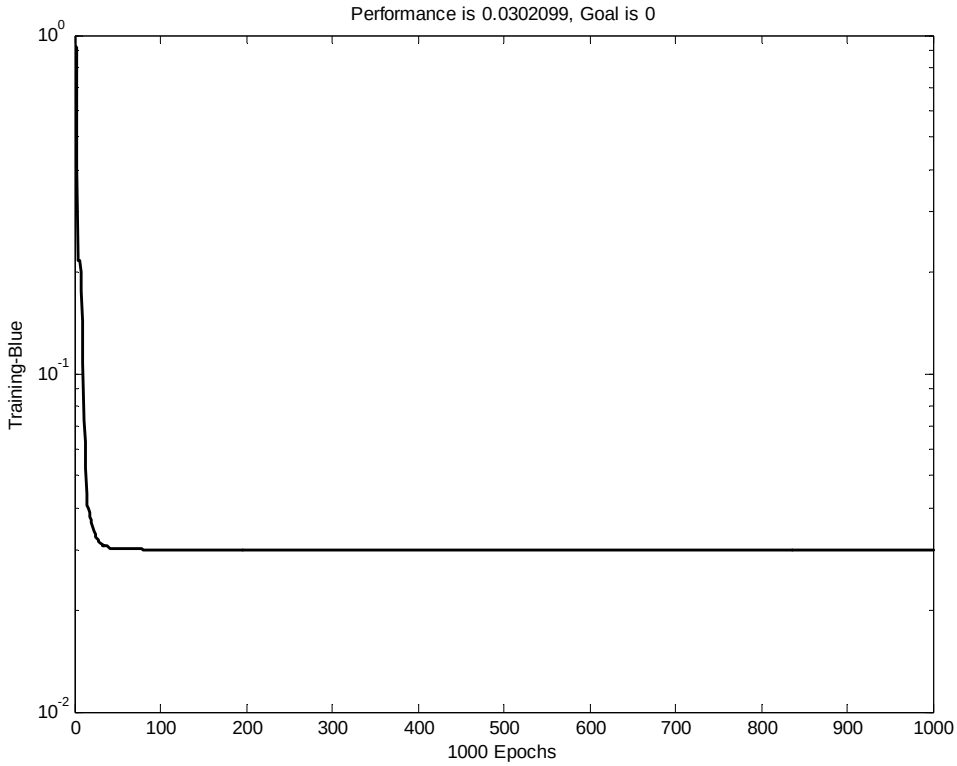

Fig. B.3 Training curve of the ANN

Appendix C

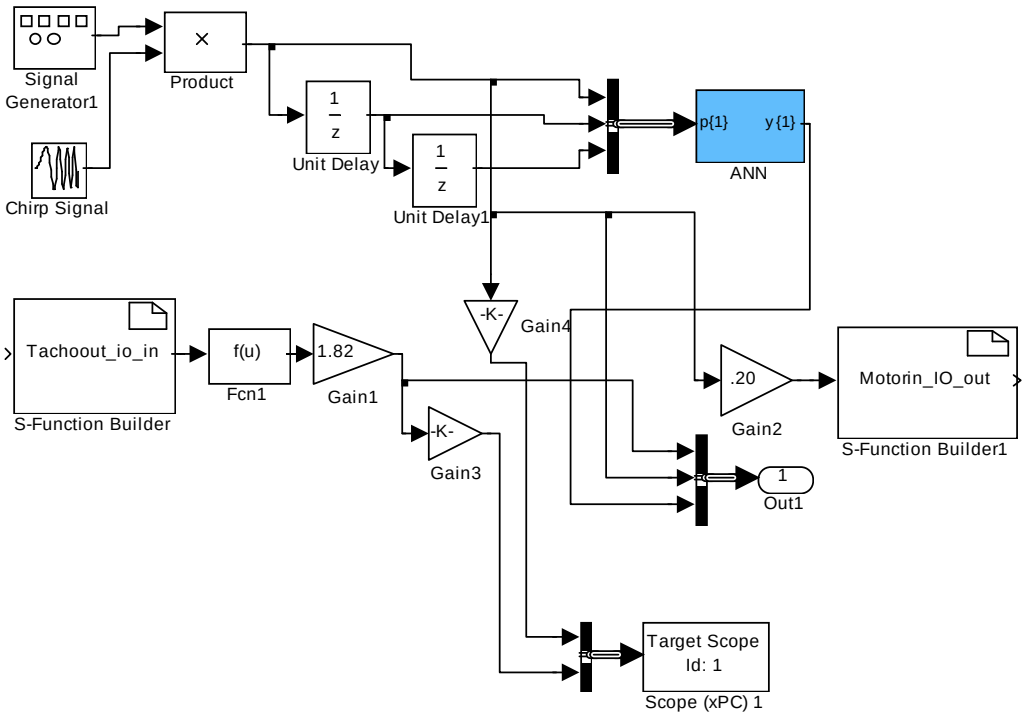

Fig C.1

Real-Time Workshop code generation for above Simulink model

```
 * Model Version                  : 1.72
 * Real-Time Workshop file version : 5.0
$Date: 2002/05/30 19:21:33 $
 * Real-Time Workshop file generated on : Wed
January  21 14:32:50 2003
 * TLC version                    : 5.0 (Jun 18
2002)
 * C source code generated on     : Wed
January 21 14:32:51 2003
 */

#include <math.h>
#include <string.h>
#include "xpc_test_Ch51.h"
#include "xpc_test_Ch51_private.h"
#include "ext_work.h"

#include "xpc_test_Ch51_dt.h"

#include "mdl_info.h"

#include "xpc_test_Ch51_bio.c"
#include "xpc_test_Ch51_pt.c"
#include "simstruc.h"

/* Block signals (auto storage) */
BlockIO rtB;

/* Block states (auto storage) */
D_Work rtDWork;

/* External output (root outports fed by signals
with auto storage) */
ExternalOutputs rtY;

/* Parent Simstruct */
static SimStruct model_S;
SimStruct *const rtS = &model_S;

real_T xpc_test_Ch51_RGND = 0.0;        /*
real_T ground */

/* Initial conditions for root system: '<Root>' */
void MdlInitialize(void)
```

```
{

  /* UnitDelay Block: <Root>/Unit Delay */
  rtDWork.Unit_Delay_DSTATE                =
rtP.Unit_Delay_X0;

  /* UnitDelay Block: <Root>/Unit Delay1 */
  rtDWork.Unit_Delay1_DSTATE               =
rtP.Unit_Delay1_X0;
}

/* Start for root system: '<Root>' */
void MdlStart(void)
{

  /*    S-Function    Block:    <S3>/S-Function
(scblock) */
  {
    int i;
    int              (XPCCALLCONV              *
rl32eScopeExists)(int ScopeNo);
    void             (XPCCALLCONV              *
rl32eDefScope)(int ScopeNo, int ScopeType);
    void             (XPCCALLCONV              *
rl32eDefTargetScope)(int ScopeNo);
    void             (XPCCALLCONV              *
rl32eAddSignal)(int ScopeNo, int SignalNo);
    void             (XPCCALLCONV              *
rl32eAddSignalTargetScope)(int ScopeNo, int
SignalNo);
    void (XPCCALLCONV * rl32eSetScope)(int
ScopeNo, int action, double value);
    void             (XPCCALLCONV              *
rl32eSetTargetScope)(int ScopeNo, int action,
double
     value);
    void             (XPCCALLCONV              *
rl32eRestartAcquisition)(int ScopeNo);

    rl32eScopeExists          =          (void*)
GetProcAddress(GetModuleHandle(NULL),
      "rl32eScopeExists");
    if (rl32eScopeExists==NULL) {
      printf("Error\n");
      return;
    }
    rl32eDefScope             =             (void*)
GetProcAddress(GetModuleHandle(NULL),
      "rl32eDefScope");
    if (rl32eDefScope==NULL) {
      printf("Error\n");
      return;
    }
    rl32eDefTargetScope          =          (void*)
GetProcAddress(GetModuleHandle(NULL),
      "rl32eDefTargetScope");
    if (rl32eDefTargetScope==NULL) {
      //printf("Error\n");
      //return;
    }
    rl32eAddSignal            =            (void*)
GetProcAddress(GetModuleHandle(NULL),
      "rl32eAddSignal");
    if (rl32eAddSignal==NULL) {
      printf("Error\n");
      return;
    }
    rl32eAddSignalTargetScope     =     (void*)
GetProcAddress(GetModuleHandle(NULL),
      "rl32eAddSignalTargetScope");
    if (rl32eAddSignalTargetScope==NULL) {
      //printf("Error\n");
      //return;
    }
    rl32eSetScope             =             (void*)
GetProcAddress(GetModuleHandle(NULL),
      "rl32eSetScope");
    if (rl32eSetScope==NULL) {
      printf("Error\n");
      return;
    }
    rl32eSetTargetScope          =          (void*)
GetProcAddress(GetModuleHandle(NULL),
      "rl32eSetTargetScope");
    if (rl32eSetTargetScope==NULL) {
      //printf("Error\n");
      //return;
    }
    rl32eRestartAcquisition       =       (void*)
GetProcAddress(GetModuleHandle(NULL),
      "rl32eRestartAcquisition");
    if (rl32eRestartAcquisition==NULL) {
      printf("Error\n");
      return;
    }

    if (!rl32eScopeExists(1)) {

      rl32eDefScope(1,2);
      rl32eDefTargetScope(1);
      rl32eAddSignal(1,
       60);
      rl32eAddSignalTargetScope(1,60);
      rl32eAddSignal(1,
       59);
      rl32eAddSignalTargetScope(1,59);
      rl32eSetScope(1, 4, 10);
      rl32eSetScope(1, 40, 0);
      rl32eSetScope(1, 7, 1);
      rl32eSetScope(1, 0, 0);
      rl32eSetScope(1, 3,
       60);
      rl32eSetScope(1, 1, 0.0);
      rl32eSetScope(1, 2, 0);
      rl32eSetScope(1, 8, -1);
      rl32eSetScope(1, 10, 0);
      rl32eSetTargetScope(1, 1, 3.0);
      rl32eSetTargetScope(1, 11, 0.0);
      rl32eSetTargetScope(1, 10, 0.0);
      rl32eRestartAcquisition(1);
```

```
   }
  }

  /* UnitDelay Block: <Root>/Unit Delay */
  rtB.Unit_Delay = rtP.Unit_Delay_X0;

  /* UnitDelay Block: <Root>/Unit Delay1 */
  rtB.Unit_Delay1 = rtP.Unit_Delay1_X0;

  MdlInitialize();
}

/* Outputs for root system: '<Root>' */
void MdlOutputs(int_T tid)
{
  /* tid is required for a uniform function
interface. This system
   * is single rate, and in this case, tid is not
accessed. */
  UNUSED_PARAMETER(tid);

  /*    S-Function    "Tachoout_io_in_wrapper"
Block: <Root>/S-Function Builder */


Tachoout_io_in_Outputs_wrapper(&xpc_test_
Ch51_RGND, &rtB.S_Function_Builder);

  /* Fcn: '<Root>/Fcn1'
   *
   * Regarding '<Root>/Fcn1':
   *   Expression: (u[1]-32768)/32768
   */
  rtB.Fcn1 = ( rtB.S_Function_Builder -
32768.0) / 32768.0;

  /* Gain: '<Root>/Gain1'
   *
   * Regarding '<Root>/Gain1':
   *   Gain value: rtP.Gain1_a_Gain
   */
  rtB.Gain1_a = rtB.Fcn1 * rtP.Gain1_a_Gain;

  /*        SignalGenerator:        '<Root>/Signal
Generator1' */
  {
    real_T sin2PiFT =

sin(6.2831853071795862E+000*rtP.Signal_Ge
nerator1_Frequency*ssGetT(rtS));
    rtB.Signal_Generator1                    =
rtP.Signal_Generator1_Amplitude*sin2PiFT;
  }

  /* Clock: '<S2>/Clock1' */
  rtB.Clock1 = ssGetT(rtS);

  /* Product: '<S2>/Product' incorporates:
   *   Constant: '<S2>/deltaFreq'
   *   Constant: '<S2>/targetTime'
   */
  rtB.Product_a = rtP.deltaFreq_Value /
rtP.targetTime_Value;

  /* Gain: '<S2>/Gain'
   *
   * Regarding '<S2>/Gain':
   *   Gain value: rtP.Gain_a_Gain
   */
  rtB.Gain_a = rtB.Product_a *
rtP.Gain_a_Gain;

  /* Product: '<S2>/Product1' */
  rtB.Product1 = rtB.Clock1 * rtB.Gain_a;

  /* Sum: '<S2>/Sum' incorporates:
   *   Constant: '<S2>/initialFreq'
   */
  rtB.Sum_a = rtB.Product1 +
rtP.initialFreq_Value;

  /* Product: '<S2>/Product2' */
  rtB.Product2 = rtB.Clock1 * rtB.Sum_a;

  /*    Trigonometry:    '<S2>/Trigonometric
Function' */
  rtB.Trigonometric_Function                 =
sin(rtB.Product2);

  /* Product: '<Root>/Product' */
  rtB.Product_b = rtB.Signal_Generator1 *
rtB.Trigonometric_Function;

  /* UnitDelay: '<Root>/Unit Delay' */
  rtB.Unit_Delay                             =
rtDWork.Unit_Delay_DSTATE;

  /* UnitDelay: '<Root>/Unit Delay1' */
  rtB.Unit_Delay1                            =
rtDWork.Unit_Delay1_DSTATE;

  /* Product: '<S12>/Product' incorporates:
   *   Constant: '<S7>/IW{1,1}(1,:)"
   */
  rtB.Product_c[0] = rtP.IW_1_1_1_Value[0] *
rtB.Product_b;
  rtB.Product_c[1] = rtP.IW_1_1_1_Value[1] *
rtB.Unit_Delay;
  rtB.Product_c[2] = rtP.IW_1_1_1_Value[2] *
rtB.Unit_Delay1;

  /* Sum: '<S12>/Sum' */
  rtB.Sum_b = rtB.Product_c[0];
  rtB.Sum_b += rtB.Product_c[1];
  rtB.Sum_b += rtB.Product_c[2];

  /* Product: '<S13>/Product' incorporates:
   *   Constant: '<S7>/IW{1,1}(2,:)"
   */
```

```
  rtB.Product_d[0] = rtP.IW_1_1_2_Value[0] *
rtB.Product_b;
  rtB.Product_d[1] = rtP.IW_1_1_2_Value[1] *
rtB.Unit_Delay;
  rtB.Product_d[2] = rtP.IW_1_1_2_Value[2] *
rtB.Unit_Delay1;

  /* Sum: '<S13>/Sum' */
  rtB.Sum_c = rtB.Product_d[0];
  rtB.Sum_c += rtB.Product_d[1];
  rtB.Sum_c += rtB.Product_d[2];

  /* Product: '<S14>/Product' incorporates:
   *   Constant: '<S7>/IW{1,1}(3,:)"
   */
  rtB.Product_e[0] = rtP.IW_1_1_3_Value[0] *
rtB.Product_b;
  rtB.Product_e[1] = rtP.IW_1_1_3_Value[1] *
rtB.Unit_Delay;
  rtB.Product_e[2] = rtP.IW_1_1_3_Value[2] *
rtB.Unit_Delay1;

  /* Sum: '<S14>/Sum' */
  rtB.Sum_d = rtB.Product_e[0];
  rtB.Sum_d += rtB.Product_e[1];
  rtB.Sum_d += rtB.Product_e[2];

  /* Sum: '<S4>/netsum' incorporates:
   *   Constant: '<S4>/b{1}'
   */
  rtB.netsum_a[0]      =      rtB.Sum_b      +
rtP.b_1_Value[0];
  rtB.netsum_a[1]      =      rtB.Sum_c      +
rtP.b_1_Value[1];
  rtB.netsum_a[2]      =      rtB.Sum_d      +
rtP.b_1_Value[2];

  /* Gain: '<S8>/Gain'
   *
   * Regarding '<S8>/Gain':
   *   Gain value: rtP.Gain_b_Gain
   */
  rtB.Gain_b[0]      =      rtB.netsum_a[0]      *
rtP.Gain_b_Gain;
  rtB.Gain_b[1]      =      rtB.netsum_a[1]      *
rtP.Gain_b_Gain;
  rtB.Gain_b[2]      =      rtB.netsum_a[2]      *
rtP.Gain_b_Gain;

  /* ElementaryMath: '<S8>/Exp' */
  rtB.Exp_a[0] = exp(rtB.Gain_b[0]);
  rtB.Exp_a[1] = exp(rtB.Gain_b[1]);
  rtB.Exp_a[2] = exp(rtB.Gain_b[2]);

  /* Sum: '<S8>/Sum' incorporates:
   *   Constant: '<S8>/one'
   */
  rtB.Sum_e[0]      =      rtB.Exp_a[0]      +
rtP.one_a_Value;
  rtB.Sum_e[1]      =      rtB.Exp_a[1]      +
rtP.one_a_Value;
  rtB.Sum_e[2]      =      rtB.Exp_a[2]      +
rtP.one_a_Value;

  /* ElementaryMath: '<S8>/Reciprocal' */
  rtB.Reciprocal_a[0] = 1.0/(rtB.Sum_e[0]);
  rtB.Reciprocal_a[1] = 1.0/(rtB.Sum_e[1]);
  rtB.Reciprocal_a[2] = 1.0/(rtB.Sum_e[2]);

  /* Gain: '<S8>/Gain1'
   *
   * Regarding '<S8>/Gain1':
   *   Gain value: rtP.Gain1_b_Gain
   */
  rtB.Gain1_b[0]      =      rtB.Reciprocal_a[0]      *
rtP.Gain1_b_Gain;
  rtB.Gain1_b[1]      =      rtB.Reciprocal_a[1]      *
rtP.Gain1_b_Gain;
  rtB.Gain1_b[2]      =      rtB.Reciprocal_a[2]      *
rtP.Gain1_b_Gain;

  /* Sum: '<S8>/Sum1' incorporates:
   *   Constant: '<S8>/one1'
   */
  rtB.Sum1_a[0]      =      rtB.Gain1_b[0]      -
rtP.one1_a_Value;
  rtB.Sum1_a[1]      =      rtB.Gain1_b[1]      -
rtP.one1_a_Value;
  rtB.Sum1_a[2]      =      rtB.Gain1_b[2]      -
rtP.one1_a_Value;

  /* Product: '<S19>/Product' incorporates:
   *   Constant: '<S16>/IW{2,1}(1,:)"
   */
  rtB.Product_f[0] = rtP.IW_2_1_1_Value[0] *
rtB.Sum1_a[0];
  rtB.Product_f[1] = rtP.IW_2_1_1_Value[1] *
rtB.Sum1_a[1];
  rtB.Product_f[2] = rtP.IW_2_1_1_Value[2] *
rtB.Sum1_a[2];

  /* Sum: '<S19>/Sum' */
  rtB.Sum_f = rtB.Product_f[0];
  rtB.Sum_f += rtB.Product_f[1];
  rtB.Sum_f += rtB.Product_f[2];

  /* Sum: '<S5>/netsum' incorporates:
   *   Constant: '<S5>/b{2}'
   */
  rtB.netsum_b = rtB.Sum_f + rtP.b_2_Value;

  /* Gain: '<S17>/Gain'
   *
   * Regarding '<S17>/Gain':
   *   Gain value: rtP.Gain_c_Gain
   */
  rtB.Gain_c      =      rtB.netsum_b      *
rtP.Gain_c_Gain;
```

```
  /* ElementaryMath: '<S17>/Exp' */
  rtB.Exp_b = exp(rtB.Gain_c);

  /* Sum: '<S17>/Sum' incorporates:
   *   Constant: '<S17>/one'
   */
  rtB.Sum_g = rtB.Exp_b + rtP.one_b_Value;

  /* ElementaryMath: '<S17>/Reciprocal' */
  rtB.Reciprocal_b = 1.0/(rtB.Sum_g);

  /* Gain: '<S17>/Gain1'
   *
   * Regarding '<S17>/Gain1':
   *   Gain value: rtP.Gain1_c_Gain
   */
  rtB.Gain1_c = rtB.Reciprocal_b *
rtP.Gain1_c_Gain;

  /* Sum: '<S17>/Sum1' incorporates:
   *   Constant: '<S17>/one1'
   */
  rtB.Sum1_b = rtB.Gain1_c -
rtP.one1_b_Value;

  /* Outport: '<Root>/Out1' */
  rtY.Out1[0] = rtB.Gain1_a;
  rtY.Out1[1] = rtB.Product_b;
  rtY.Out1[2] = rtB.Sum1_b;

  /* Gain: '<Root>/Gain2'
   *
   * Regarding '<Root>/Gain2':
   *   Gain value: rtP.Gain2_Gain
   */
  rtB.Gain2 = rtB.Product_b * rtP.Gain2_Gain;

  /* Gain: '<Root>/Gain3'
   *
   * Regarding '<Root>/Gain3':
   *   Gain value: rtP.Gain3_Gain
   */
  rtB.Gain3 = rtB.Gain1_a * rtP.Gain3_Gain;

  /* Gain: '<Root>/Gain4'
   *
   * Regarding '<Root>/Gain4':
   *   Gain value: rtP.Gain4_Gain
   */
  rtB.Gain4 = rtB.Product_b * rtP.Gain4_Gain;

  /*    S-Function    "Motorin_IO_out_wrapper"
Block: <Root>/S-Function Builder1 */


Motorin_IO_out_Outputs_wrapper(&rtB.Gain2
, &rtB.S_Function_Builder1);
}

/* Update for root system: '<Root>' */
void MdlUpdate(int_T tid)
{

  /* tid is required for a uniform function
interface. This system
   * is single rate, and in this case, tid is not
accessed. */
  UNUSED_PARAMETER(tid);

  /* UnitDelay Block: <Root>/Unit Delay */
  rtDWork.Unit_Delay_DSTATE =
rtB.Product_b;

  /* UnitDelay Block: <Root>/Unit Delay1 */
  rtDWork.Unit_Delay1_DSTATE =
rtB.Unit_Delay;
}

/* Terminate for root system: '<Root>' */
void MdlTerminate(void)
{
  if(rtS != NULL) {
  }
}

/* Function to initialize sizes */
void MdlInitializeSizes(void)
{
  ssSetNumContStates(rtS, 0);          /* Number
of continuous states */
  ssSetNumY(rtS, 3);                   /* Number of
model outputs */
  ssSetNumU(rtS, 0);                   /* Number of
model inputs */
  ssSetDirectFeedThrough(rtS, 0);      /* The
model is not direct feedthrough */
  ssSetNumSampleTimes(rtS, 2);         /*
Number of sample times */
  ssSetNumBlocks(rtS, 55);             /* Number
of blocks */
  ssSetNumBlockIO(rtS, 40);            /* Number
of block outputs */
  ssSetNumBlockParams(rtS, 36);        /* Sum
of parameter "widths" */
}

/* Function to initialize sample times */
void MdlInitializeSampleTimes(void)
{
  /* task periods */
  ssSetSampleTime(rtS, 0, 0.0);
  ssSetSampleTime(rtS, 1, 0.0025);

  /* task offsets */
  ssSetOffsetTime(rtS, 0, 0.0);
  ssSetOffsetTime(rtS, 1, 0.0);
}

/* Function to register the model */
SimStruct *xpc_test_Ch51(void)
```

```
{
  static struct _ssMdlInfo mdlInfo;
  (void)memset((char *)rtS, 0,
sizeof(SimStruct));
  (void)memset((char *)&mdlInfo, 0,
sizeof(struct _ssMdlInfo));
  ssSetMdlInfoPtr(rtS, &mdlInfo);

  /* timing info */
  {
    static time_T
mdlPeriod[NSAMPLE_TIMES];
    static time_T
mdlOffset[NSAMPLE_TIMES];
    static time_T
mdlTaskTimes[NSAMPLE_TIMES];
    static int_T mdlTsMap[NSAMPLE_TIMES];
    static int_T
mdlSampleHits[NSAMPLE_TIMES];

    {
      int_T i;

      for(i = 0; i < NSAMPLE_TIMES; i++) {
        mdlPeriod[i] = 0.0;
        mdlOffset[i] = 0.0;
        mdlTaskTimes[i] = 0.0;
      }
    }
    (void)memset((char_T *)&mdlTsMap[0], 0,
2 * sizeof(int_T));
    (void)memset((char_T
*)&mdlSampleHits[0], 0, 2 * sizeof(int_T));

    ssSetSampleTimePtr(rtS, &mdlPeriod[0]);
    ssSetOffsetTimePtr(rtS, &mdlOffset[0]);
    ssSetSampleTimeTaskIDPtr(rtS,
&mdlTsMap[0]);
    ssSetTPtr(rtS, &mdlTaskTimes[0]);
    ssSetSampleHitPtr(rtS, &mdlSampleHits[0]);
  }
  ssSetSolverMode(rtS,
SOLVER_MODE_SINGLETASKING);

  /*
   * initialize model vectors and cache them in
SimStruct
   */

  /* block I/O */
  {
    void *b = (void *) &rtB;
    ssSetBlockIO(rtS, b);

    {
      int_T i;

      b =&rtB.S_Function_Builder;
      for (i = 0; i < 62; i++) {
        ((real_T*)b)[i] = 0.0;
      }
    }
  }

  /* external outputs */
  {
    ssSetY(rtS, &rtY);

    {
      int_T i;

      for (i = 0; i < 3; i++) {
        rtY.Out1[i] = 0.0;
      }
    }
  }

  /* parameters */
  ssSetDefaultParam(rtS, (real_T *) &rtP);

  /* data type work */
  {
    void *dwork = (void *) &rtDWork;
    ssSetRootDWork(rtS, dwork);
    {
      int_T i;
      real_T *dwork_ptr = (real_T *)
&rtDWork.Unit_Delay_DSTATE;

      for (i = 0; i < 2; i++) {
        dwork_ptr[i] = 0.0;
      }
    }
  }

  /* data type transition information (for
external mode) */
  {
    static DataTypeTransInfo dtInfo;

    (void)memset((char_T *) &dtInfo, 0,
sizeof(dtInfo));
    ssSetModelMappingInfo(rtS, &dtInfo);

    _ssSetReservedForXPC(rtS, (void*)
&dtInfo);
    dtInfo.numDataTypes = 13;
    dtInfo.dataTypeSizes =
&rtDataTypeSizes[0];
    dtInfo.dataTypeNames =
&rtDataTypeNames[0];

    /* Block I/O transition table */
    dtInfo.B = &rtBTransTable;

    /* Parameters transition table */
    dtInfo.P = &rtPTransTable;
  }
```

```
  /* C API for Parameter Tuning and/or Signal
Monitoring */
  {
    static ModelMappingInfo mapInfo;

    memset((char_T *) &mapInfo, 0,
sizeof(mapInfo));

    /* block signal monitoring map */
    mapInfo.Signals.blockIOSignals =
&rtBIOSignals[0];
    mapInfo.Signals.numBlockIOSignals = 40;

    /* parameter tuning maps */
    mapInfo.Parameters.blockTuning =
&rtBlockTuning[0];
    mapInfo.Parameters.variableTuning =
&rtVariableTuning[0];
    mapInfo.Parameters.parametersMap =
rtParametersMap;
    mapInfo.Parameters.dimensionsMap =
rtDimensionsMap;
    mapInfo.Parameters.numBlockTuning = 26;
    mapInfo.Parameters.numVariableTuning = 0;

    ssSetModelMappingInfo(rtS, &mapInfo);
  }

  /* Model specific registration */
  ssSetRootSS(rtS, rtS);

  ssSetVersion(rtS,
SIMSTRUCT_VERSION_LEVEL2);
  ssSetModelName(rtS, "xpc_test_Ch51");
  ssSetPath(rtS, "xpc_test_Ch51");

  ssSetTStart(rtS, 0.0);
  ssSetTFinal(rtS, 50.0);
  ssSetStepSize(rtS, 0.0025);
  ssSetFixedStepSize(rtS, 0.0025);
  /* Setup for data logging */
  {
    static RTWLogInfo rt_DataLoggingInfo;

    ssSetRTWLogInfo(rtS,
&rt_DataLoggingInfo);

    rtliSetLogFormat(ssGetRTWLogInfo(rtS),
0);

    rtliSetLogMaxRows(ssGetRTWLogInfo(rtS),
1000);


rtliSetLogDecimation(ssGetRTWLogInfo(rtS),
1);


rtliSetLogVarNameModifier(ssGetRTWLogInf
o(rtS), "rt_");

    rtliSetLogT(ssGetRTWLogInfo(rtS), "tout");

    rtliSetLogX(ssGetRTWLogInfo(rtS), "");

    rtliSetLogXFinal(ssGetRTWLogInfo(rtS),
"");


rtliSetLogXSignalInfo(ssGetRTWLogInfo(rtS),
NULL);


rtliSetLogXSignalPtrs(ssGetRTWLogInfo(rtS),
NULL);

    rtliSetLogY(ssGetRTWLogInfo(rtS),
"yout");

    /*
     * Set pointers to the data and signal info for
each output
     */
    {
      static void * rt_LoggedOutputSignalPtrs[] =
{
        &rtY.Out1[0]
      };


rtliSetLogYSignalPtrs(ssGetRTWLogInfo(rtS),

((LogSignalPtrsType)rt_LoggedOutputSignalPt
rs));
    }
    {
      static const int_T rt_LoggedOutputWidths[]
= {
        3
      };

      static const int_T
rt_LoggedOutputNumDimensions[] = {
        1
      };

      static const int_T
rt_LoggedOutputDimensions[] = {
        3
      };

      static const BuiltInDTypeId
rt_LoggedOutputDataTypeIds[] = {
        SS_DOUBLE
      };

      static const int_T
rt_LoggedOutputComplexSignals[] = {
        0
      };
```

```
    static const char_T
rt_LoggedOutputLabels[] = "";

    static const int_T
rt_LoggedOutputLabelLengths[] = {
      0
    };

    static const char_T
rt_LoggedOutputBlockNames[] =
"xpc_test_Ch51/Out1";

    static const int_T
rt_LoggedOutputBlockNameLengths[] = {
      18
    };

    static const RTWLogSignalInfo
rt_LoggedOutputSignalInfo = {
      1,
      rt_LoggedOutputWidths,
      rt_LoggedOutputNumDimensions,
      rt_LoggedOutputDimensions,
      rt_LoggedOutputDataTypeIds,
      rt_LoggedOutputComplexSignals,
      NULL,
      rt_LoggedOutputLabels,
      rt_LoggedOutputLabelLengths,
      NULL,
      NULL,
      NULL,
      rt_LoggedOutputBlockNames,
      rt_LoggedOutputBlockNameLengths
    };


rtliSetLogYSignalInfo(ssGetRTWLogInfo(rtS),
&rt_LoggedOutputSignalInfo);
   }
  }

  ssSetChecksumVal(rtS, 0, 2312101917U);
  ssSetChecksumVal(rtS, 1, 2891184062U);
  ssSetChecksumVal(rtS, 2, 3346049207U);
  ssSetChecksumVal(rtS, 3, 619288640U);

  {
   static const EnableStates rtAlwaysEnabled =
SUBSYS_ENABLED;

   static RTWExtModeInfo rt_ExtModeInfo;
   static const void *sysModes[1];

   ssSetRTWExtModeInfo(rtS,
&rt_ExtModeInfo);

rteiSetSubSystemModeVectorAddresses(&rt_E
xtModeInfo, sysModes);
   sysModes[0] = &rtAlwaysEnabled;


rteiSetModelMappingInfoPtr(&rt_ExtModeInfo
, &ssGetModelMappingInfo(rtS));

   rteiSetChecksumsPtr(&rt_ExtModeInfo,
ssGetChecksums(rtS));

   rteiSetTPtr(&rt_ExtModeInfo,
ssGetTPtr(rtS));
  }

  return rtS;
}
```

Appendix D

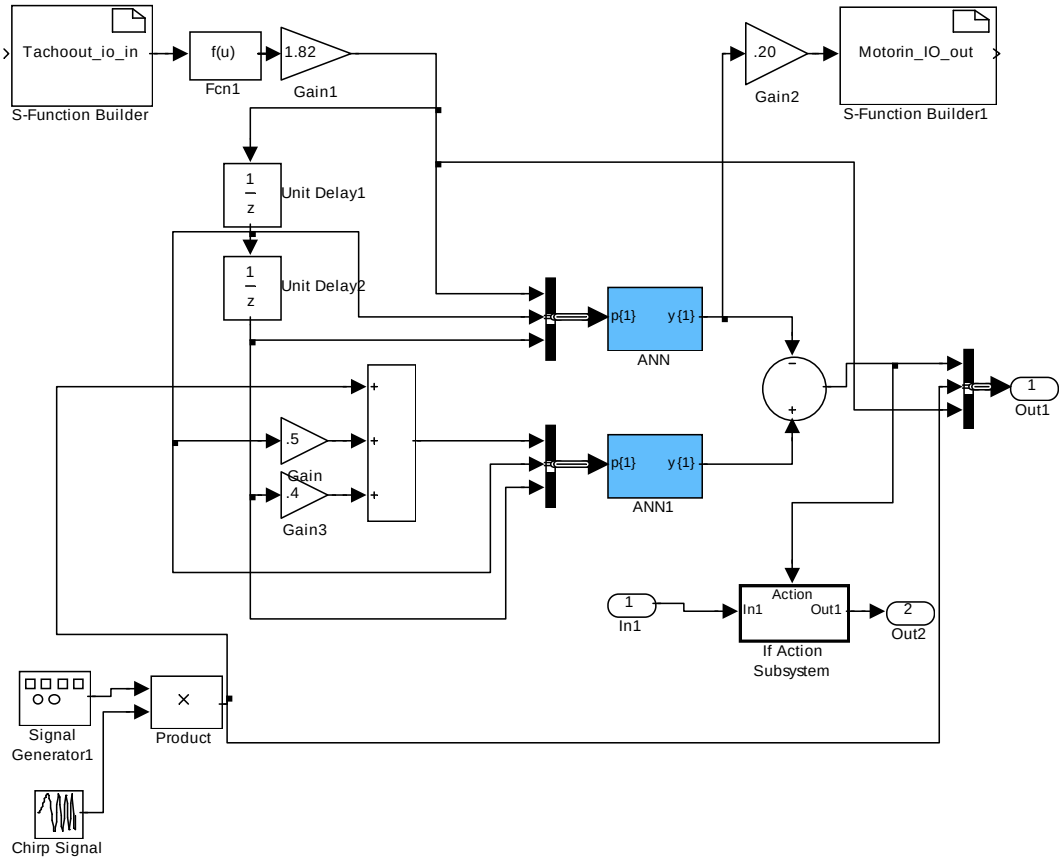

Fig D.1
Real-Time Workshop code generation for above Simulink model

```
 *
 * Model Version                  : 1.75
 * Real-Time Workshop file version      : 5.0
$Date: 2002/05/30 19:21:33 $
 * Real-Time Workshop file generated on :
Wed January  22 16:36:29 2003
 * TLC version                    : 5.0 (Jun 18
2002)
 * C source code generated on           : Thu
May 22 16:36:29 2003
 */

#include <math.h>
#include <string.h>
#include "xpc_test_Ch52.h"
#include "xpc_test_Ch52_private.h"
#include "ext_work.h"

#include "xpc_test_Ch52_dt.h"

#include "mdl_info.h"

#include "xpc_test_Ch52_bio.c"

#include "xpc_test_Ch52_pt.c"
#include "simstruc.h"

/* Block signals (auto storage) */
BlockIO rtB;

/* Block states (auto storage) */
D_Work rtDWork;
```

```
/* External inputs (root inport signals with auto
storage) */
ExternalInputs rtU;

/* External output (root outports fed by signals
with auto storage) */
ExternalOutputs rtY;

/* Parent Simstruct */
static SimStruct model_S;
SimStruct *const rtS = &model_S;

real_T xpc_test_Ch52_RGND = 0.0;        /*
real_T ground */

/* Initial conditions for root system: '<Root>'
*/
void MdlInitialize(void)
{

  /* UnitDelay Block: <Root>/Unit Delay1 */
  rtDWork.Unit_Delay1_DSTATE =
rtP.Unit_Delay1_X0;

  /* UnitDelay Block: <Root>/Unit Delay2 */
  rtDWork.Unit_Delay2_DSTATE =
rtP.Unit_Delay2_X0;
}

/* Start for root system: '<Root>' */
void MdlStart(void)
{

  /* UnitDelay Block: <Root>/Unit Delay1 */
  rtB.Unit_Delay1 = rtP.Unit_Delay1_X0;

  /* UnitDelay Block: <Root>/Unit Delay2 */
  rtB.Unit_Delay2 = rtP.Unit_Delay2_X0;

  MdlInitialize();
}

/* Outputs for root system: '<Root>' */
void MdlOutputs(int_T tid)
{
  /* tid is required for a uniform function
interface. This system
   * is single rate, and in this case, tid is not
accessed. */
  UNUSED_PARAMETER(tid);

  /* S-Function "Tachoout_io_in_wrapper"
Block: <Root>/S-Function Builder */


Tachoout_io_in_Outputs_wrapper(&xpc_test_
Ch52_RGND, &rtB.S_Function_Builder);

  /* Fcn: '<Root>/Fcn1'
   *
   * Regarding '<Root>/Fcn1':
   *   Expression: (u[1]-32768)/32768
   */
  rtB.Fcn1 = ( rtB.S_Function_Builder -
32768.0) / 32768.0;

  /* Gain: '<Root>/Gain1'
   *
   * Regarding '<Root>/Gain1':
   *   Gain value: rtP.Gain1_a_Gain
   */
  rtB.Gain1_a = rtB.Fcn1 * rtP.Gain1_a_Gain;

  /* UnitDelay: '<Root>/Unit Delay1' */
  rtB.Unit_Delay1 =
rtDWork.Unit_Delay1_DSTATE;

  /* UnitDelay: '<Root>/Unit Delay2' */
  rtB.Unit_Delay2 =
rtDWork.Unit_Delay2_DSTATE;

  /* Product: '<S13>/Product' incorporates:
   *   Constant: '<S8>/IW{1,1}(1,:)"
   */
  rtB.Product_a[0] =
rtP.IW_1_1_1_a_Value[0] * rtB.Gain1_a;
  rtB.Product_a[1] =
rtP.IW_1_1_1_a_Value[1] * rtB.Unit_Delay1;
  rtB.Product_a[2] =
rtP.IW_1_1_1_a_Value[2] * rtB.Unit_Delay2;

  /* Sum: '<S13>/Sum' */
  rtB.Sum_a = rtB.Product_a[0];
  rtB.Sum_a += rtB.Product_a[1];
  rtB.Sum_a += rtB.Product_a[2];

  /* Product: '<S14>/Product' incorporates:
   *   Constant: '<S8>/IW{1,1}(2,:)"
   */
  rtB.Product_b[0] =
rtP.IW_1_1_2_a_Value[0] * rtB.Gain1_a;
  rtB.Product_b[1] =
rtP.IW_1_1_2_a_Value[1] * rtB.Unit_Delay1;
  rtB.Product_b[2] =
rtP.IW_1_1_2_a_Value[2] * rtB.Unit_Delay2;

  /* Sum: '<S14>/Sum' */
  rtB.Sum_b = rtB.Product_b[0];
  rtB.Sum_b += rtB.Product_b[1];
  rtB.Sum_b += rtB.Product_b[2];

  /* Product: '<S15>/Product' incorporates:
   *   Constant: '<S8>/IW{1,1}(3,:)"
   */
  rtB.Product_c[0] =
rtP.IW_1_1_3_a_Value[0] * rtB.Gain1_a;
  rtB.Product_c[1] =
rtP.IW_1_1_3_a_Value[1] * rtB.Unit_Delay1;
  rtB.Product_c[2] =
rtP.IW_1_1_3_a_Value[2] * rtB.Unit_Delay2;
```

```
  /* Sum: '<S15>/Sum' */
  rtB.Sum_c = rtB.Product_c[0];
  rtB.Sum_c += rtB.Product_c[1];
  rtB.Sum_c += rtB.Product_c[2];

  /* Sum: '<S5>/netsum' incorporates:
   *   Constant: '<S5>/b{1}'
   */
  rtB.netsum_a[0] = rtB.Sum_a +
rtP.b_1_a_Value[0];
  rtB.netsum_a[1] = rtB.Sum_b +
rtP.b_1_a_Value[1];
  rtB.netsum_a[2] = rtB.Sum_c +
rtP.b_1_a_Value[2];

  /* Gain: '<S9>/Gain'
   *
   * Regarding '<S9>/Gain':
   *   Gain value: rtP.Gain_a_Gain
   */
  rtB.Gain_a[0] = rtB.netsum_a[0] *
rtP.Gain_a_Gain;
  rtB.Gain_a[1] = rtB.netsum_a[1] *
rtP.Gain_a_Gain;
  rtB.Gain_a[2] = rtB.netsum_a[2] *
rtP.Gain_a_Gain;

  /* ElementaryMath: '<S9>/Exp' */
  rtB.Exp_a[0] = exp(rtB.Gain_a[0]);
  rtB.Exp_a[1] = exp(rtB.Gain_a[1]);
  rtB.Exp_a[2] = exp(rtB.Gain_a[2]);

  /* Sum: '<S9>/Sum' incorporates:
   *   Constant: '<S9>/one'
   */
  rtB.Sum_d[0] = rtB.Exp_a[0] +
rtP.one_a_Value;
  rtB.Sum_d[1] = rtB.Exp_a[1] +
rtP.one_a_Value;
  rtB.Sum_d[2] = rtB.Exp_a[2] +
rtP.one_a_Value;

  /* ElementaryMath: '<S9>/Reciprocal' */
  rtB.Reciprocal_a[0] = 1.0/(rtB.Sum_d[0]);
  rtB.Reciprocal_a[1] = 1.0/(rtB.Sum_d[1]);
  rtB.Reciprocal_a[2] = 1.0/(rtB.Sum_d[2]);

  /* Gain: '<S9>/Gain1'
   *
   * Regarding '<S9>/Gain1':
   *   Gain value: rtP.Gain1_b_Gain
   */
  rtB.Gain1_b[0] = rtB.Reciprocal_a[0] *
rtP.Gain1_b_Gain;
  rtB.Gain1_b[1] = rtB.Reciprocal_a[1] *
rtP.Gain1_b_Gain;
  rtB.Gain1_b[2] = rtB.Reciprocal_a[2] *
rtP.Gain1_b_Gain;

  /* Sum: '<S9>/Sum1' incorporates:
   *   Constant: '<S9>/one1'
   */
  rtB.Sum1_a[0] = rtB.Gain1_b[0] -
rtP.one1_a_Value;
  rtB.Sum1_a[1] = rtB.Gain1_b[1] -
rtP.one1_a_Value;
  rtB.Sum1_a[2] = rtB.Gain1_b[2] -
rtP.one1_a_Value;

  /* Product: '<S20>/Product' incorporates:
   *   Constant: '<S17>/IW{2,1}(1,:)"
   */
  rtB.Product_d[0] =
rtP.IW_2_1_1_a_Value[0] * rtB.Sum1_a[0];
  rtB.Product_d[1] =
rtP.IW_2_1_1_a_Value[1] * rtB.Sum1_a[1];
  rtB.Product_d[2] =
rtP.IW_2_1_1_a_Value[2] * rtB.Sum1_a[2];

  /* Sum: '<S20>/Sum' */
  rtB.Sum_e = rtB.Product_d[0];
  rtB.Sum_e += rtB.Product_d[1];
  rtB.Sum_e += rtB.Product_d[2];

  /* Sum: '<S6>/netsum' incorporates:
   *   Constant: '<S6>/b{2}'
   */
  rtB.netsum_b = rtB.Sum_e +
rtP.b_2_a_Value;

  /* Gain: '<S18>/Gain'
   *
   * Regarding '<S18>/Gain':
   *   Gain value: rtP.Gain_b_Gain
   */
  rtB.Gain_b = rtB.netsum_b *
rtP.Gain_b_Gain;

  /* ElementaryMath: '<S18>/Exp' */
  rtB.Exp_b = exp(rtB.Gain_b);

  /* Sum: '<S18>/Sum' incorporates:
   *   Constant: '<S18>/one'
   */
  rtB.Sum_f = rtB.Exp_b + rtP.one_b_Value;

  /* ElementaryMath: '<S18>/Reciprocal' */
  rtB.Reciprocal_b = 1.0/(rtB.Sum_f);

  /* Gain: '<S18>/Gain1'
   *
   * Regarding '<S18>/Gain1':
   *   Gain value: rtP.Gain1_c_Gain
   */
  rtB.Gain1_c = rtB.Reciprocal_b *
rtP.Gain1_c_Gain;

  /* Sum: '<S18>/Sum1' incorporates:
   *   Constant: '<S18>/one1'
```

```
   */
  rtB.Sum1_b = rtB.Gain1_c -
rtP.one1_b_Value;

  /* SignalGenerator: '<Root>/Signal
Generator1' */
  {
    real_T sin2PiFT =

sin(6.2831853071795862E+000*rtP.Signal_G
enerator1_Frequency*ssGetT(rtS));
    rtB.Signal_Generator1 =
rtP.Signal_Generator1_Amplitude*sin2PiFT;
  }

  /* Clock: '<S3>/Clock1' */
  rtB.Clock1 = ssGetT(rtS);

  /* Product: '<S3>/Product' incorporates:
   *   Constant: '<S3>/deltaFreq'
   *   Constant: '<S3>/targetTime'
   */
  rtB.Product_e = rtP.deltaFreq_Value /
rtP.targetTime_Value;

  /* Gain: '<S3>/Gain'
   *
   * Regarding '<S3>/Gain':
   *   Gain value: rtP.Gain_c_Gain
   */
  rtB.Gain_c = rtB.Product_e *
rtP.Gain_c_Gain;

  /* Product: '<S3>/Product1' */
  rtB.Product1 = rtB.Clock1 * rtB.Gain_c;

  /* Sum: '<S3>/Sum' incorporates:
   *   Constant: '<S3>/initialFreq'
   */
  rtB.Sum_g = rtB.Product1 +
rtP.initialFreq_Value;

  /* Product: '<S3>/Product2' */
  rtB.Product2 = rtB.Clock1 * rtB.Sum_g;

  /* Trigonometry: '<S3>/Trigonometric
Function' */
  rtB.Trigonometric_Function =
sin(rtB.Product2);

  /* Product: '<Root>/Product' */
  rtB.Product_f = rtB.Signal_Generator1 *
rtB.Trigonometric_Function;

  /* Gain: '<Root>/Gain'
   *
   * Regarding '<Root>/Gain':
   *   Gain value: rtP.Gain_d_Gain
   */
  rtB.Gain_d = rtB.Unit_Delay1 *
rtP.Gain_d_Gain;

  /* Gain: '<Root>/Gain3'
   *
   * Regarding '<Root>/Gain3':
   *   Gain value: rtP.Gain3_Gain
   */
  rtB.Gain3 = rtB.Unit_Delay2 *
rtP.Gain3_Gain;

  /* Sum: '<Root>/Sum' */
  rtB.Sum_h = rtB.Product_f + rtB.Gain_d +
rtB.Gain3;

  /* Product: '<S29>/Product' incorporates:
   *   Constant: '<S24>/IW{1,1}(1,:)"
   */
  rtB.Product_g[0] =
rtP.IW_1_1_1_b_Value[0] * rtB.Sum_h;
  rtB.Product_g[1] =
rtP.IW_1_1_1_b_Value[1] * rtB.Unit_Delay1;
  rtB.Product_g[2] =
rtP.IW_1_1_1_b_Value[2] * rtB.Unit_Delay2;

  /* Sum: '<S29>/Sum' */
  rtB.Sum_i = rtB.Product_g[0];
  rtB.Sum_i += rtB.Product_g[1];
  rtB.Sum_i += rtB.Product_g[2];

  /* Product: '<S30>/Product' incorporates:
   *   Constant: '<S24>/IW{1,1}(2,:)"
   */
  rtB.Product_h[0] =
rtP.IW_1_1_2_b_Value[0] * rtB.Sum_h;
  rtB.Product_h[1] =
rtP.IW_1_1_2_b_Value[1] * rtB.Unit_Delay1;
  rtB.Product_h[2] =
rtP.IW_1_1_2_b_Value[2] * rtB.Unit_Delay2;

  /* Sum: '<S30>/Sum' */
  rtB.Sum_j = rtB.Product_h[0];
  rtB.Sum_j += rtB.Product_h[1];
  rtB.Sum_j += rtB.Product_h[2];

  /* Product: '<S31>/Product' incorporates:
   *   Constant: '<S24>/IW{1,1}(3,:)"
   */
  rtB.Product_i[0] = rtP.IW_1_1_3_b_Value[0]
* rtB.Sum_h;
  rtB.Product_i[1] = rtP.IW_1_1_3_b_Value[1]
* rtB.Unit_Delay1;
  rtB.Product_i[2] = rtP.IW_1_1_3_b_Value[2]
* rtB.Unit_Delay2;

  /* Sum: '<S31>/Sum' */
  rtB.Sum_k = rtB.Product_i[0];
  rtB.Sum_k += rtB.Product_i[1];
  rtB.Sum_k += rtB.Product_i[2];
```

```
  /* Sum: '<S21>/netsum' incorporates:
   *   Constant: '<S21>/b{1}'
   */
  rtB.netsum_c[0] = rtB.Sum_i +
rtP.b_1_b_Value[0];
  rtB.netsum_c[1] = rtB.Sum_j +
rtP.b_1_b_Value[1];
  rtB.netsum_c[2] = rtB.Sum_k +
rtP.b_1_b_Value[2];

  /* Gain: '<S25>/Gain'
   *
   * Regarding '<S25>/Gain':
   *   Gain value: rtP.Gain_e_Gain
   */
  rtB.Gain_e[0] = rtB.netsum_c[0] *
rtP.Gain_e_Gain;
  rtB.Gain_e[1] = rtB.netsum_c[1] *
rtP.Gain_e_Gain;
  rtB.Gain_e[2] = rtB.netsum_c[2] *
rtP.Gain_e_Gain;

  /* ElementaryMath: '<S25>/Exp' */
  rtB.Exp_c[0] = exp(rtB.Gain_e[0]);
  rtB.Exp_c[1] = exp(rtB.Gain_e[1]);
  rtB.Exp_c[2] = exp(rtB.Gain_e[2]);

  /* Sum: '<S25>/Sum' incorporates:
   *   Constant: '<S25>/one'
   */
  rtB.Sum_l[0] = rtB.Exp_c[0] +
rtP.one_c_Value;
  rtB.Sum_l[1] = rtB.Exp_c[1] +
rtP.one_c_Value;
  rtB.Sum_l[2] = rtB.Exp_c[2] +
rtP.one_c_Value;

  /* ElementaryMath: '<S25>/Reciprocal' */
  rtB.Reciprocal_c[0] = 1.0/(rtB.Sum_l[0]);
  rtB.Reciprocal_c[1] = 1.0/(rtB.Sum_l[1]);
  rtB.Reciprocal_c[2] = 1.0/(rtB.Sum_l[2]);

  /* Gain: '<S25>/Gain1'
   *
   * Regarding '<S25>/Gain1':
   *   Gain value: rtP.Gain1_d_Gain
   */
  rtB.Gain1_d[0] = rtB.Reciprocal_c[0] *
rtP.Gain1_d_Gain;
  rtB.Gain1_d[1] = rtB.Reciprocal_c[1] *
rtP.Gain1_d_Gain;
  rtB.Gain1_d[2] = rtB.Reciprocal_c[2] *
rtP.Gain1_d_Gain;

  /* Sum: '<S25>/Sum1' incorporates:
   *   Constant: '<S25>/one1'
   */
  rtB.Sum1_c[0] = rtB.Gain1_d[0] -
rtP.one1_c_Value;
  rtB.Sum1_c[1] = rtB.Gain1_d[1] -
rtP.one1_c_Value;
  rtB.Sum1_c[2] = rtB.Gain1_d[2] -
rtP.one1_c_Value;

  /* Product: '<S36>/Product' incorporates:
   *   Constant: '<S33>/IW{2,1}(1,:)"
   */
  rtB.Product_j[0] = rtP.IW_2_1_1_b_Value[0]
* rtB.Sum1_c[0];
  rtB.Product_j[1] = rtP.IW_2_1_1_b_Value[1]
* rtB.Sum1_c[1];
  rtB.Product_j[2] = rtP.IW_2_1_1_b_Value[2]
* rtB.Sum1_c[2];

  /* Sum: '<S36>/Sum' */
  rtB.Sum_m = rtB.Product_j[0];
  rtB.Sum_m += rtB.Product_j[1];
  rtB.Sum_m += rtB.Product_j[2];

  /* Sum: '<S22>/netsum' incorporates:
   *   Constant: '<S22>/b{2}'
   */
  rtB.netsum_d = rtB.Sum_m +
rtP.b_2_b_Value;

  /* Gain: '<S34>/Gain'
   *
   * Regarding '<S34>/Gain':
   *   Gain value: rtP.Gain_f_Gain
   */
  rtB.Gain_f = rtB.netsum_d *
rtP.Gain_f_Gain;

  /* ElementaryMath: '<S34>/Exp' */
  rtB.Exp_d = exp(rtB.Gain_f);

  /* Sum: '<S34>/Sum' incorporates:
   *   Constant: '<S34>/one'
   */
  rtB.Sum_n = rtB.Exp_d + rtP.one_d_Value;

  /* ElementaryMath: '<S34>/Reciprocal' */
  rtB.Reciprocal_d = 1.0/(rtB.Sum_n);

  /* Gain: '<S34>/Gain1'
   *
   * Regarding '<S34>/Gain1':
   *   Gain value: rtP.Gain1_e_Gain
   */
  rtB.Gain1_e = rtB.Reciprocal_d *
rtP.Gain1_e_Gain;

  /* Sum: '<S34>/Sum1' incorporates:
   *   Constant: '<S34>/one1'
   */
  rtB.Sum1_d = rtB.Gain1_e -
rtP.one1_d_Value;

  /* Sum: '<Root>/Sum1' */
```

```
  rtB.Sum1_e = - rtB.Sum1_b + rtB.Sum1_d;

  /* Outport: '<Root>/Out1' */
  rtY.Out1[0] = rtB.Sum1_e;
  rtY.Out1[1] = rtB.Product_f;
  rtY.Out1[2] = rtB.Gain1_a;

  /* Gain: '<Root>/Gain2'
   *
   * Regarding '<Root>/Gain2':
   *   Gain value: rtP.Gain2_Gain
   */
  rtB.Gain2 = rtB.Sum1_b * rtP.Gain2_Gain;

  /* S-Function "Motorin_IO_out_wrapper"
Block: <Root>/S-Function Builder1 */


Motorin_IO_out_Outputs_wrapper(&rtB.Gain
2, &rtB.S_Function_Builder1);

  /* If: '<Root>/TmpIf_Feeding_If Action
Subsystem_AtInput2' */
  if (0.0 !=0.0) {                  /* u1 ~= 0 */

    /* Output and update for action system:
'<Root>/If Action Subsystem' */

    /* Inport: '<S4>/In1' incorporates:
     *   Inport: '<Root>/In1'
     */
    rtB.In1 = rtU.In1;
  }

  /* Outport: '<Root>/Out2' */
  rtY.Out2 = rtB.In1;
}

/* Update for root system: '<Root>' */
void MdlUpdate(int_T tid)
{

  /* tid is required for a uniform function
interface. This system
   * is single rate, and in this case, tid is not
accessed. */
  UNUSED_PARAMETER(tid);

  /* UnitDelay Block: <Root>/Unit Delay1 */
  rtDWork.Unit_Delay1_DSTATE =
rtB.Gain1_a;

  /* UnitDelay Block: <Root>/Unit Delay2 */
  rtDWork.Unit_Delay2_DSTATE =
rtB.Unit_Delay1;
}

/* Terminate for root system: '<Root>' */
void MdlTerminate(void)
{
  if(rtS != NULL) {
  }
}

/* Function to initialize sizes */
void MdlInitializeSizes(void)
{
  ssSetNumContStates(rtS, 0);          /*
Number of continuous states */
  ssSetNumY(rtS, 4);                   /* Number of
model outputs */
  ssSetNumU(rtS, 1);                   /* Number of
model inputs */
  ssSetDirectFeedThrough(rtS, 1);      /* The
model is direct feedthrough */
  ssSetNumSampleTimes(rtS, 2);         /*
Number of sample times */
  ssSetNumBlocks(rtS, 92);             /* Number
of blocks */
  ssSetNumBlockIO(rtS, 65);            /*
Number of block outputs */
  ssSetNumBlockParams(rtS, 60);        /* Sum
of parameter "widths" */
}

/* Function to initialize sample times */
void MdlInitializeSampleTimes(void)
{
  /* task periods */
  ssSetSampleTime(rtS, 0, 0.0);
  ssSetSampleTime(rtS, 1, 0.0025);

  /* task offsets */
  ssSetOffsetTime(rtS, 0, 0.0);
  ssSetOffsetTime(rtS, 1, 0.0);
}

/* Function to register the model */
SimStruct *xpc_test_Ch52(void)
{
  static struct _ssMdlInfo mdlInfo;
  (void)memset((char *)rtS, 0,
sizeof(SimStruct));
  (void)memset((char *)&mdlInfo, 0,
sizeof(struct _ssMdlInfo));
  ssSetMdlInfoPtr(rtS, &mdlInfo);

  /* timing info */
  {
    static time_T
mdlPeriod[NSAMPLE_TIMES];
    static time_T
mdlOffset[NSAMPLE_TIMES];
    static time_T
mdlTaskTimes[NSAMPLE_TIMES];
    static int_T
mdlTsMap[NSAMPLE_TIMES];
    static int_T
mdlSampleHits[NSAMPLE_TIMES];
```

```
  {
    int_T i;

    for(i = 0; i < NSAMPLE_TIMES; i++) {
      mdlPeriod[i] = 0.0;
      mdlOffset[i] = 0.0;
      mdlTaskTimes[i] = 0.0;
    }
  }
  (void)memset((char_T *)&mdlTsMap[0], 0,
2 * sizeof(int_T));
  (void)memset((char_T
*)&mdlSampleHits[0], 0, 2 * sizeof(int_T));

  ssSetSampleTimePtr(rtS, &mdlPeriod[0]);
  ssSetOffsetTimePtr(rtS, &mdlOffset[0]);
  ssSetSampleTimeTaskIDPtr(rtS,
&mdlTsMap[0]);
  ssSetTPtr(rtS, &mdlTaskTimes[0]);
  ssSetSampleHitPtr(rtS,
&mdlSampleHits[0]);
 }
 ssSetSolverMode(rtS,
SOLVER_MODE_SINGLETASKING);

 /*
  * initialize model vectors and cache them in
SimStruct
  */

 /* block I/O */
 {
  void *b = (void *) &rtB;
  ssSetBlockIO(rtS, b);

  {
    int_T i;

    b = &rtB.S_Function_Builder;
    for (i = 0; i < 108; i++) {
      ((real_T*)b)[i] = 0.0;
    }
    b =&rtB.In1;
    for (i = 0; i < 1; i++) {
      ((real_T*)b)[i] = 0.0;
    }
  }
 }

 /* external inputs */
 {
  ssSetU(rtS, ((void*) &rtU));

  rtU.In1 = 0.0;
 }

 /* external outputs */
 {
  ssSetY(rtS, &rtY);

  {
    int_T i;

    for (i = 0; i < 3; i++) {
      rtY.Out1[i] = 0.0;
    }
  }
  rtY.Out2 = 0.0;
 }

 /* parameters */
 ssSetDefaultParam(rtS, (real_T *) &rtP);

 /* data type work */
 {
  void *dwork = (void *) &rtDWork;
  ssSetRootDWork(rtS, dwork);
  {
    int_T i;
    real_T *dwork_ptr = (real_T *)
&rtDWork.Unit_Delay1_DSTATE;

    for (i = 0; i < 2; i++) {
      dwork_ptr[i] = 0.0;
    }
  }
 }

 /* data type transition information (for
external mode) */
 {
  static DataTypeTransInfo dtInfo;

  (void)memset((char_T *) &dtInfo, 0,
sizeof(dtInfo));
  ssSetModelMappingInfo(rtS, &dtInfo);

  _ssSetReservedForXPC(rtS, (void*)
&dtInfo);
  dtInfo.numDataTypes = 13;
  dtInfo.dataTypeSizes =
&rtDataTypeSizes[0];
  dtInfo.dataTypeNames =
&rtDataTypeNames[0];

  /* Block I/O transition table */
  dtInfo.B = &rtBTransTable;

  /* Parameters transition table */
  dtInfo.P = &rtPTransTable;
 }

 /* C API for Parameter Tuning and/or Signal
Monitoring */
 {
  static ModelMappingInfo mapInfo;

  memset((char_T *) &mapInfo, 0,
sizeof(mapInfo));
```

```
    /* block signal monitoring map */
    mapInfo.Signals.blockIOSignals =
&rtBIOSignals[0];
    mapInfo.Signals.numBlockIOSignals = 65;

    /* parameter tuning maps */
    mapInfo.Parameters.blockTuning =
&rtBlockTuning[0];
    mapInfo.Parameters.variableTuning =
&rtVariableTuning[0];
    mapInfo.Parameters.parametersMap =
rtParametersMap;
    mapInfo.Parameters.dimensionsMap =
rtDimensionsMap;
    mapInfo.Parameters.numBlockTuning = 40;
    mapInfo.Parameters.numVariableTuning =
0;

    ssSetModelMappingInfo(rtS, &mapInfo);
  }

  /* Model specific registration */
  ssSetRootSS(rtS, rtS);

  ssSetVersion(rtS,
SIMSTRUCT_VERSION_LEVEL2);
  ssSetModelName(rtS, "xpc_test_Ch52");
  ssSetPath(rtS, "xpc_test_Ch52");

  ssSetTStart(rtS, 0.0);
  ssSetTFinal(rtS, 50.0);
  ssSetStepSize(rtS, 0.0025);
  ssSetFixedStepSize(rtS, 0.0025);
  /* Setup for data logging */
  {
    static RTWLogInfo rt_DataLoggingInfo;

    ssSetRTWLogInfo(rtS,
&rt_DataLoggingInfo);

    rtliSetLogFormat(ssGetRTWLogInfo(rtS),
0);


rtliSetLogMaxRows(ssGetRTWLogInfo(rtS),
1000);


rtliSetLogDecimation(ssGetRTWLogInfo(rtS),
1);


rtliSetLogVarNameModifier(ssGetRTWLogIn
fo(rtS), "rt_");

    rtliSetLogT(ssGetRTWLogInfo(rtS),
"tout");

    rtliSetLogX(ssGetRTWLogInfo(rtS), "");

    rtliSetLogXFinal(ssGetRTWLogInfo(rtS),
"");


rtliSetLogXSignalInfo(ssGetRTWLogInfo(rtS)
, NULL);


rtliSetLogXSignalPtrs(ssGetRTWLogInfo(rtS)
, NULL);

    rtliSetLogY(ssGetRTWLogInfo(rtS),
"yout");

    /*
     * Set pointers to the data and signal info for
each output
     */
    {
      static void * rt_LoggedOutputSignalPtrs[]
= {
        &rtY.Out1[0],
        &rtY.Out2
      };


rtliSetLogYSignalPtrs(ssGetRTWLogInfo(rtS)
,

((LogSignalPtrsType)rt_LoggedOutputSignalP
trs));
    }
    {
      static const int_T
rt_LoggedOutputWidths[] = {
        3,
        1
      };

      static const int_T
rt_LoggedOutputNumDimensions[] = {
        1,
        1
      };

      static const int_T
rt_LoggedOutputDimensions[] = {
        3,
        1
      };

      static const BuiltInDTypeId
rt_LoggedOutputDataTypeIds[] = {
        SS_DOUBLE,
        SS_DOUBLE
      };

      static const int_T
rt_LoggedOutputComplexSignals[] = {
        0,
```

```
      0
     };

     static const char_T
rt_LoggedOutputLabels[] = ""
      "";

     static const int_T
rt_LoggedOutputLabelLengths[] = {
      0,
      0
     };

     static const char_T
rt_LoggedOutputBlockNames[] =
"xpc_test_Ch52/Out1"
      "xpc_test_Ch52/Out2";

     static const int_T
rt_LoggedOutputBlockNameLengths[] = {
      18,
      18
     };

     static const RTWLogSignalInfo
rt_LoggedOutputSignalInfo = {
      2,
      rt_LoggedOutputWidths,
      rt_LoggedOutputNumDimensions,
      rt_LoggedOutputDimensions,
      rt_LoggedOutputDataTypeIds,
      rt_LoggedOutputComplexSignals,
      NULL,
      rt_LoggedOutputLabels,
      rt_LoggedOutputLabelLengths,
      NULL,
      NULL,
      NULL,
      rt_LoggedOutputBlockNames,
      rt_LoggedOutputBlockNameLengths
     };


rtliSetLogYSignalInfo(ssGetRTWLogInfo(rtS)
, &rt_LoggedOutputSignalInfo);
   }
  }

  ssSetChecksumVal(rtS, 0, 2778520006U);
  ssSetChecksumVal(rtS, 1, 805833585U);
  ssSetChecksumVal(rtS, 2, 2636373961U);
  ssSetChecksumVal(rtS, 3, 2549496657U);

  {
   static const EnableStates rtAlwaysEnabled =
SUBSYS_ENABLED;

   static RTWExtModeInfo rt_ExtModeInfo;
   static const void *sysModes[2];

   ssSetRTWExtModeInfo(rtS,
&rt_ExtModeInfo);

rteiSetSubSystemModeVectorAddresses(&rt_
ExtModeInfo, sysModes);

   sysModes[0] = &rtAlwaysEnabled;
   sysModes[1] = &rtAlwaysEnabled;


rteiSetModelMappingInfoPtr(&rt_ExtModeInf
o, &ssGetModelMappingInfo(rtS));

   rteiSetChecksumsPtr(&rt_ExtModeInfo,
ssGetChecksums(rtS));

   rteiSetTPtr(&rt_ExtModeInfo,
ssGetTPtr(rtS));
  }

  return rtS;
}
```